

Education

The Art of C++

C++ 编程艺术

(美) Herbert Schildt 著

曹蓉蓉 刘小荷 翻译
毕长剑 战晓苏 审校

Mc
Graw
Hill

清华大学出版社



The Art of C++

本书揭示了C++程序员创建世界级软件的奥秘。程序设计大师Herbert Schildt通过将C++语言广泛应用于功能强大的编程任务中，全面展示了C++语言的多功能性、敏捷性和艺术性。本书内容包括探索C++的功能，创建内存管理的垃圾回收子系统，开发线程控制面板，建立编译器以扩展C++的功能，开发可断点续传的Internet文件下载工具，创建财务分析库，用基于AI的搜索技术探索人工智能，建立定制的STL容器，以及开发Mini C++解释程序。书中所有示例和项目的源代码都可以从www.osborne.com上免费下载。

本书是Herbert Schildt的又一本精心力作，书中用生动的语言深入浅出地描述了C++语言的强大和完美！

——Ed Felten (Princeton大学教授，美国)

这是一本C++程序员盼望已久的精品书籍，能快速提升C++程序员的编程技能。

——Tony Scott (IS&S首席技术执行官，美国)

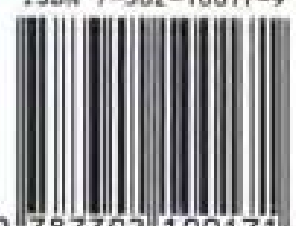
不要让编程工作成为一种机械劳动！本书指导您在C++编程实践中展现自己的艺术和优雅！

——Elinor Mills Abreu (Reuter专栏作者，英国)

作者以C++国际标准语法为基础，详细、生动地讲解C++语言编程技术，并结合多年软件开发和教学经验总结出非常有价值的完整示例，以行之有效的方法让读者精通C++语言编程。

——战晓苏(教授，计算机专家，中国)

ISBN 7-302-10017-9



9 787302 100171 >

定价：39.80 元



McGraw-Hill
全球智慧中文化

<http://www.mheducation.com>

C++编程艺术

(美) Herbert Schildt 著

曹蓉蓉 刘小荷 翻译

毕长剑 战晓苏 审校

清华大学出版社

北 京

Herbert Schildt
The Art of C++
EISBN: 0-07-225512-9

Copyright © 2004 by The McGraw-Hill Companies, Inc.

Original language published by The McGraw-Hill Companies, Inc. All Rights reserved. No part of this publication may be reproduced or distributed by any means, or stored in a database or retrieval system, without the prior written permission of the publisher.

Simplified Chinese translation edition is published and distributed exclusively by Tsinghua University Press under the authorization by McGraw-Hill Education(Asia) Co., within the territory of the People's Republic of China only (excluding Hong Kong, Macao SAR and Taiwan). Unauthorized export of this edition is a violation of the Copyright Act. Violation of this Law is subject to Civil and Criminal Penalties.

本书中文简体字翻译版由美国麦格劳-希尔教育出版(亚洲)公司授权清华大学出版社在中华人民共和国境内(不包括中国香港、澳门特别行政区和中国台湾地区)独家出版发行。未经许可之出口视为违反著作权法,将受法律之制裁。未经出版者预先书面许可,不得以任何方式复制或抄袭本书的任何部分。

北京市版权局著作权合同登记号 图字:01-2004-3726

版权所有,翻印必究。举报电话:010-62782989 13501256678 13801310933

本书封面贴有 McGraw-Hill 公司防伪标签,无标签者不得销售。

图书在版编目(CIP)数据

C++编程艺术/(美)斯切尔特(Schildt, H.)著;曹蓉蓉 刘小荷翻译. —北京:清华大学出版社, 2005.4

书名原文: The Art of C++

ISBN 7-302-10017-9

I. C… II. ①斯…②曹…③刘… III. C语言—程序设计 IV.TP312

中国版本图书馆 CIP 数据核字(2004)第 124674 号

出版者:清华大学出版社

<http://www.tup.com.cn>

社总机:010-62770175

组稿编辑:曹 康

封面设计:康 博

印刷者:北京市清华园胶印厂

发行者:新华书店总店北京发行所

开 本:185×260 印 张:21.25 字 数:544千字

版 次:2005年4月第1版 2005年4月第1次印刷

书 号:ISBN 7-302-10017-9/TP·6875

印 数:1~6000

定 价:39.80元

地 址:北京清华大学学研大厦

邮 编:100084

客户服务:010-62776969

文稿编辑:丁 平

版式设计:康 博

装 订 者:三河市金元装订厂

译者序

本书作者 Herbert Schildt, 是公认的 C、C++、Java 和 C# 等主流编程语言的程序设计大师和 Windows 程序设计专家, 也是顶级编程图书作者; 同时, 他还是 ANSI/ISO 和 C++ 标准化组织的专家。他的编程书籍被翻译成多种语言版本广为流传, 在全球世界范围内的销量已经超过三百万册。

本书的突出特点之一是编程技巧全面。本书以 C++ 国际标准语法为基础, 从高级特性全面讲解 C++ 语言编程技术、技巧, 充分展示了 C++ 语言的强大性、多样性、优美性、敏捷性和艺术性。作者结合多年软件开发和教学经验总结出非常有价值的完整示例, 以行之有效的方法让读者快速精通 C++ 语言编程技巧。本书每章给出的示例代码都可以直接运行, 无需修改, 而且读者可以从 www.osborne.com 网站免费下载本书代码。相信读者通过研读本书可大大提高 C++ 编程能力。

本书的突出特点之二是内容丰富实用。在本书中, Herbert Schildt 给出了许多非常实用的高性能程序, 每个程序分别侧重于 C++ 语言的不同方面。本书的实用示例程序主要包括垃圾回收器子系统、可以断点续传的 Internet 文件下载程序、线程控制面板、基于人工智能(AI)的搜索程序、通用 STL 容器和小型 C++ 解释程序等。作者对每个程序都给出了非常细致的分析和解释, 这对于读者来说无疑是大有帮助的。本书代码示例易懂有趣、设计思想独特, 从中读者可以学到很多在其他 C++ 书中无法学到的技巧, 使读者能够掌握 C++ 高级编程的技巧, 真正进入 C++ 高级编程领域, 值得每位 C++ 程序员阅读和珍藏。

本书的突出特点之三是文笔透彻精确。Herbert Schildt 是全球著名的程序设计语言书籍作者, 本书秉承他一贯的写作风格: 简捷、清晰, 非常适合读者的学习和阅读。通过本书的阅读, 读者可在 C++ 程序设计大师的引领下探索编写高性能 C++ 程序的奥秘。本书让读者的 C++ 编程技术更上一层楼。

本书为 C++ 编程高级读物, 面向有初级 C++ 语言基础和一定编程经历的程序设计者, 适合作为高等院校计算机专业相关课程辅助教材, 也可作为高年级本科生、研究生和广大编程爱好者深入学习 C++ 及其他面向对象语言的技术参考书。

本书由曹蓉蓉、刘小荷翻译; 毕长剑、战晓苏审校。虽然与 Herbert Schildt 的高超造诣相比甚远, 但是本书的译者和审校者还是凭借多年的 C++ 教学和编程经验, 在翻译、校译此书的过程中, 抱着对读者认真负责的态度, 力争将原书的风格和思想原原本本地呈现给读者。还要特别指出的是, 清华大学出版社对本书非常重视, 从全书的翻译、编辑、排版到印刷质量上都下了很大的功夫。因此, 可以相信这本中译本能够成为对读者大有裨益的好书。

此外, 我们曾专门以电子邮件方式与原书作者进行了多次沟通, 将原书的个别错误在翻译、校译和审校过程中加以改正。限于水平的原因, 中译本中不妥或错误之处在所难免, 敬请广大读者批评指正。我们的 e-mail 是 fwkbook@tup.tsinghua.edu.cn, 读者有问题可随时联系。

前言

从早期的 FORTRAN 语言开始，计算机语言就一直不断地发展演变。在此过程中，消除了不稳固的特性，取而代之的是一些功能强大的特性。多年过去后，这些进化的努力被精练为一种纯粹的形式，那就是程序设计语言应该具有的纯净本质。这么多年努力的结果就是 C++ 语言的产生，在程序设计历史上，任何其他语言都没有像 C++ 语言拥有如此重要的地位。

C++ 的成功取决于许多原因。语法简洁而优雅；对象模型简明流畅，容易理解；C++ 中提供了精心编写的库。然而，并非是这些特性为 C++ 赢得历史上的重要地位，而是 C++ 给予程序设计人员的强大的功能。从来没有其他的语言能够使得程序设计人员更加直接地控制计算机。通过使用 C++，程序设计人员就是机器的主人——这正是所有的程序设计人员所需要的。

没有边界，没有限制，没有约束。这就是 C++ 语言。

0.1 本书内容

本书不同于大多数其他的 C++ 书籍。其他的 C++ 书籍讲授语言的基础，而本书展示了如何应用 C++ 在更大的范围内完成有趣的、有用的、甚至是神秘的程序设计任务。在此过程中充分显示了 C++ 语言的强大功能和优雅性。

大致来说，本书包含两类应用程序，第一类称为“纯代码”，因为它们注重于扩展 C++ 程序设计环境本身。第 2 章的垃圾回收器，第 3 章的线程控制面板以及第 8 章的定制 STL 容器都是这种类型的示例。第二类应用程序显示了如何应用 C++ 来完成各种计算任务。例如，第 5 章开发了一个可以断点续传的网络下载工具，第 6 章给出了一个如何建立财务应用程序的范例，第 8 章中应用 C++ 实现了人工智能应用。

本书以一段独特有趣的代码结束：Mini C++ 解释程序，这个程序可以解释 C++ 的一个小子集。Mini C++ 解释程序揭示了 C++ 的关键字和语法是如何一起工作从而组成这门语言的语法的。更重要的是，这可以使您了解这门语言的内部机制，并且显示了隐藏在 C++ 设计背后的一些原因。使用 Mini C++ 解释程序不仅有趣，它还可以用作开发您自己的语言的起点，还可以用作其他语言的解释程序。

本书每一章都提供了可以直接使用的代码。例如，第 2 章的垃圾回收器可以适用于许多程序设计任务。然而，只有把这些应用程序作为自己的开发起点，才会获益匪浅。例如，可以进一步完善第 8 章的 Internet 文件下载工具，使其可以在某个指定的时间开始下载，或者可以监控一个下载站点，保持下载最新的文件。总之，可以将这些不同的程序和子系统作为您开发自己项目的跳板。

0.2 预备知识

本书假定读者具有坚实的C++语言基础知识。读者应该能够创建、编译并运行C++程序。应该能够使用指针、模板以及异常处理，理解复制构造函数并且熟悉标准库的常用部分。因此，本书假定读者具有可以从C++教程中获得的技巧。

如果读者需要复习或者加强基础知识，作者推荐下面几本书。

C++ From the Ground Up

C++: A Beginner's Guide

C++: The Complete Reference

这3本书都是 McGraw-Hill/Osborne 出版社发行的。其中 *C++: A Beginner's Guide* 一书的第1版和第2版已经由清华大学出版社出版发行，书名为《C++基础教程》和《C++基础教程(第2版)》。

0.3 源代码

本书所有的示例和项目的源代码都可以从网站 www.osborne.com 上免费下载。

目 录

第 1 章 C++的功能	1
1.1 简洁而丰富的语法	1
1.2 功能强大的库	2
1.3 STL	2
1.4 程序员控制一切	3
1.5 细节控制	3
1.6 运算符重载	3
1.7 一种简洁精练的对象模型	4
1.8 C++发展史	4
第 2 章 简单的 C++垃圾回收器	5
2.1 两种内存管理方法的比较	5
2.1.1 手工内存管理的优缺点	6
2.1.2 垃圾回收的优缺点	6
2.1.3 两种方法都可以使用	7
2.2 在 C++中创建垃圾回收器	7
2.3 选择垃圾回收的算法	8
2.3.1 引用计数	9
2.3.2 标记并清除	9
2.3.3 复制	9
2.3.4 采用哪种算法	9
2.3.5 实现垃圾回收器	10
2.3.6 是否使用多线程	10
2.3.7 何时回收垃圾	10
2.3.8 关于 auto_ptr	11
2.4 一个简单的 C++垃圾回收器	11
2.5 详细讨论 GCPtr	23
2.5.1 GCPtr 的数据成员	23
2.5.2 函数 findPtrInfo()	24
2.5.3 GCIterator typedef	25
2.5.4 GCPtr 的构造函数	25
2.5.5 GCPtr 的析构函数	26
2.5.6 回收垃圾函数 collect()	26
2.5.7 重载赋值运算符	28
2.5.8 GCPtr 的复制构造函数	30

2.5.9	指针运算符和转换函数	30
2.5.10	begin()和end()函数	32
2.5.11	shutdown()函数	32
2.5.12	两个实用函数	33
2.6	GCInfo	33
2.7	Iter	34
2.8	如何使用 GCPtr	36
2.8.1	处理分配异常	37
2.8.2	一个更有趣的示例	38
2.8.3	对象的分配和丢弃	40
2.8.4	分配数组	41
2.8.5	使用具有类类型的 GCPtr	43
2.8.6	一个比较大的演示程序	45
2.8.7	加载测试	51
2.8.8	一些限制	53
2.9	试着完成下面的任务	53
第3章	C++中的多线程	54
3.1	什么是多线程	54
3.2	为什么 C++没有内建支持多线程	55
3.3	选用什么样的操作系统和编译器	56
3.4	Windows 线程函数概述	56
3.4.1	线程的创建和终止	56
3.4.2	Visual C++对 CreateThread()和 ExitThread()的替换	57
3.4.3	线程的挂起和恢复	58
3.4.4	改变线程的优先级	59
3.4.5	获取主线程的句柄	60
3.4.6	同步	60
3.5	创建线程控制面板	63
3.5.1	线程控制面板	64
3.5.2	线程控制面板的详细分析	68
3.5.3	控制面板的演示	74
3.6	一个多线程的垃圾回收器	78
3.6.1	附加的成员变量	79
3.6.2	多线程的 GCPtr 构造函数	79
3.6.3	TimeOutExc 异常	81
3.6.4	多线程的 GCPtr 析构函数	81
3.6.5	gc()函数	82
3.6.6	isRunning()函数	82
3.6.7	gclist 的同步访问	83

3.6.8 其他两个改变	83
3.6.9 完整的多线程垃圾回收器	83
3.6.10 多线程垃圾回收器的使用	95
3.7 试着完成下面的任务	97
第4章 C++的扩展	98
4.1 为什么使用译码器	98
4.2 实验性的关键字	99
4.2.1 foreach 循环	99
4.2.2 cases 语句	100
4.2.3 typeof 运算符	101
4.2.4 repeat/until 循环	102
4.3 试验 C++新特性的译码器	102
4.4 使用译码器	111
4.5 译码器的运行方式	112
4.5.1 全局声明	112
4.5.2 main()函数	112
4.5.3 gettoken()和 skipspaces()函数	114
4.5.4 转换 foreach 循环	117
4.5.5 转换 cases 语句	119
4.5.6 转换 typeof 运算符	121
4.5.7 转换 repeat/until 循环	122
4.6 演示程序	124
4.7 尝试完成以下任务	130
第5章 Internet 文件下载工具	131
5.1 WinINet 库	131
5.2 文件下载工具子系统	132
5.2.1 操作的一般理论	137
5.2.2 download()函数	137
5.2.3 ishttp()函数	142
5.2.4 httpverOK()函数	142
5.2.5 getfname()函数	143
5.2.6 openfile()函数	143
5.2.7 update()函数	144
5.3 Download 头文件	145
5.4 文件下载工具的演示	145
5.5 基于 GUI 的下载工具	147
5.5.1 WinDL 代码	147
5.5.2 WinDL 的运行方式	152
5.6 尝试完成以下任务	153

第 6 章 使用 C++ 的财务计算	154
6.1 计算贷款的定期偿还	154
6.2 计算投资的预期价值	156
6.3 计算为了获得预期的价值所需的原始投资	157
6.4 为了获得预期的养老金所需的原始投资	159
6.5 计算给定投资所能得到的养老金的最大值	160
6.6 计算贷款余额	162
6.7 尝试完成以下任务	163
第 7 章 基于 AI 的问题求解	164
7.1 表示法和术语	164
7.2 组合爆炸	165
7.3 搜索方法	167
7.4 需要解决的问题	167
7.5 FlightInfo 结构和 Search 类	169
7.6 深度优先搜索	171
7.6.1 match() 函数	176
7.6.2 find() 函数	177
7.6.3 findroute() 函数	177
7.6.4 显示路线	179
7.6.5 深度优先搜索分析	179
7.7 广度优先搜索	179
7.8 添加启发信息	182
7.8.1 爬山搜索法	183
7.8.2 爬山法分析	189
7.9 最低成本搜索	189
7.10 寻找多解	190
7.10.1 路径删除	191
7.10.2 节点删除	192
7.11 寻找“最优”解决方案	198
7.12 回到丢失钥匙的问题	204
7.13 尝试完成以下任务	207
第 8 章 定制 STL 容器	208
8.1 STL 的简要回顾	208
8.1.1 容器	209
8.1.2 算法	209
8.1.3 迭代器	209
8.2 其他的 STL 实体	209

8.3	定制容器的要求	210
8.3.1	一般要求	210
8.3.2	序列式容器的其他要求	211
8.3.3	关联式容器的要求	211
8.4	创建范围可选的动态数组容器	212
8.4.1	RangeArray 的运行方式	212
8.4.2	完整的 RangeArray 类	213
8.4.3	详细讨论 RangeArray 类	224
8.4.4	一些 RangeArray 示例程序	235
8.4.5	尝试完成以下任务	245
第 9 章	Mini C++解释程序	246
9.1	解释程序和编译器	246
9.2	Mini C++纵览	247
9.3	Mini C++说明	247
9.4	非正式的 C++理论	249
9.4.1	C++表达式	250
9.4.2	定义表达式	250
9.5	表达式解析器	252
9.5.1	解析器代码	252
9.5.2	分解源代码	264
9.5.3	显示语法错误	270
9.5.4	表达式求值	271
9.6	Mini C++解释程序	272
9.6.1	main()函数	291
9.6.2	解释程序的预扫描程序	292
9.6.3	interp()函数	295
9.6.4	处理局部变量	297
9.6.5	调用用户自定义的函数	299
9.6.6	给变量赋值	300
9.6.7	执行 if 语句	302
9.6.8	switch 语句和 break 语句	304
9.6.9	处理 while 循环	306
9.6.10	处理 do-while 循环	307
9.6.11	for 循环	308
9.6.12	处理 cin 和 cout 语句	309
9.7	Mini C++的库函数	311
9.8	mccommon.h 头文件	313
9.9	编译并链接 Mini C++解释程序	315

9.10	演示 Mini C++	315
9.11	改进 Mini C++	323
9.12	扩展 Mini C++	324
9.12.1	添加新的 C++特性	324
9.12.2	添加辅助特性	325

第 1 章 C++ 的功能

C++的功能精深而广博，包括：可以直接对机器底层进行编程控制，生成高效的运行代码，具有直接与操作系统交互的能力。使用 C++，可以全面控制对象，包括对象的创建、销毁以及继承，可以访问指针，并且支持底层 I/O。可以通过定义类和重载运算符来加入新的功能。可以创建自己的库并手工优化代码。甚至可以在需要的时候“打破规则”。C++并不是为谨小慎微者准备的语言，它是为需要世界上最强大编程语言的程序员准备的。

当然，C++并不只是具有原始功能。它的功能是有条理的、集中的并且是直接的。它的精心设计、功能丰富的库以及微妙的语法形成了灵活的编程环境。尽管 C++的特长在于创建高性能的系统代码，但是它也可以用来完成任何其他编程任务。例如，它的字符串处理能力是其他语言无法比拟的，它的数学和数字运算能力使其特别适用于科学计算编程，C++可以生成高效的目标代码，可以很好地完成需要大量复杂运算的任务。

这本书的目标是展示 C++的功能、应用范围以及它的灵活性。我们通过用 C++实现各种不同的应用程序来做到这一点。一些应用程序说明了语言本身的能力。这些应用程序被称为“纯代码”示例，因为它们体现了 C++的语法规则以及使用 C++的方便性、灵活性和简洁性。第 2 章的垃圾回收器和第 9 章的 C++解释程序就是这样的示例。其他应用程序说明了 C++可以很容易地应用到普遍的程序设计任务中。例如，第 5 章的下载管理器体现了 C++创建高性能的网络代码的能力。第 6 章将 C++应用于各种财务计算。总之，这些应用程序显示了 C++是一种功能强大的语言。

然而，在开始接触应用程序之前，花费一点时间思考一下是什么使得 C++成为如此优秀的语言，这对您的学习是有好处的。为此，本章花一些时间来指出赋予了“C++功能”的几个特性。

1.1 简洁而丰富的语法

C++的一个主要特征是它语法的简洁。C++语言只定义了 63 个关键字。C++定义了丰富但简洁的语法来提供控制语句、运算符、数据类型以及任何现代语言所需要的面向对象的特征。因此，C++的语法是简练、一致并紧凑的。

这种够用就好的哲学有两个重要的优点。首先，C++的关键字和语法可以应用于所有可以使用 C++的环境。也就是说，C++的核心特征对于所有的应用程序都有效，而不依赖于执行环境。对于依赖于执行环境的功能，如多线程，C++交给操作系统来处理，操作系统能够有效地处理这些问题。因此，C++没有试图采用一种“万能”的解决方案，因为那样做会降低运行效率。

其次，一种改进的、逻辑上一致的语法应能够清晰地表达复杂的结构。当程序变得庞大的时候，这一点就显得尤其重要。当然，笨拙的程序员编写笨拙的 C++代码，优秀的程序员编写清晰而简明的代码。这种能清楚地表示出复杂逻辑的能力是 C++成为广泛应用的程序设计语言

的原因之一。

1.2 功能强大的库

当然，现代程序设计环境需要许多超出 C++ 关键字和语法支持的功能。C++ 通过标准库提供了这些功能。C++ 定义了任何主流语言都可以使用的设计良好的库。它的函数库由 C 语言沿袭而来，包含了范围广泛的一组非面向对象的函数，如 `char*` 字符串的处理、字符处理以及转换函数，这些函数被程序员广泛应用。C++ 类库为 I/O、字符串以及 STL 等提供面向对象的支持。

由于依赖于库例程而不是关键字，因此只需要简单地扩展库，而不是发明新关键字，就可以把新的功能加入到库中。这使得 C++ 可以适应程序设计环境的改变，而不需要改变核心语言。因此，C++ 兼顾了看起来有点冲突的特性：稳定性和灵活性。

即使在它的函数库和类库中，C++ 也采用了一种够用就好、“化繁为简”的方法，从而避免了“万能”陷阱。这个库仅提供了可以为广泛的编程环境合理实现的功能。对于适用于特定环境的特殊功能，C++ 通过操作系统提供。因此，C++ 程序员可以使用执行平台的所有功能。这种方法使您可以编写最大限度使用执行环境的属性和能力的高效代码。

1.3 STL

标准类库中有一个部分特别重要，因而有必要拿出来单独讨论：这就是 STL (Standard Template Library, 标准模板库)，STL 的创建改变了程序员思考和使用语言库的方式。其影响如此深远，以至于影响了后出现的语言的设计。例如，Java 和 C# 的 Collection 架构就是直接按照 STL 的模式创建的。

标准模板库 (STL) 的核心是一套完善的模板类与函数，它实现了许多常用的数据结构，标准模板库将其称为容器。例如，STL 包含支持向量、链表、队列和堆栈的容器。由于 STL 是由模板类和函数构成的，因此其容器几乎可用于任何数据类型。因此，STL 提供了解决各种编程问题的“即用”解决方案。

尽管 STL 对于 C++ 程序员的实际重要性不会被低估，但是仍然有一个很重要的原因使得它非常重要。它是软件组件革命的先驱。由此开始，程序员开始寻找重用代码的方法。因为开发和调试是一个代价很高的过程，所以代码重用非常符合需求。在早期，代码重用是通过将一个程序的源代码剪切并粘贴到另一个程序中完成的。(当然，现在这个方法也有效)。后来，程序员创建了可以重复使用的函数库，如 C++ 提供的那些库。紧接着就是标准类库。

在类库的基础上，STL 更进一步地采用了一个重要的概念：将库模块化为适用于多种数据的通用组件。另外，由于 STL 是可扩展的，因此您可以定义自己的容器，添加自己的算法，甚至改写内建的容器。扩展、修改以及重用功能的能力是软件组件的本质。

现在，组件软件革命已经接近尾声，很容易忘记是 STL 革命，包括模块化的功能、标准化的接口以及通过继承的可扩展性。计算的历史将把 STL 作为语言设计中重要的里程碑编入史册。

1.4 程序员控制一切

在此有两种程序设计语言的理论。一些人认为语言应该具有避免会在第一“现场”引发问题的特性，从而阻止程序员出现错误。这听起来很不错，但是这样做通常会将某些功能强大的特性被限制、削弱甚至被完全排除，虽然这些特性具有潜在的危险性。这种特性的两个示例是指针和显式内存分配。指针被认为是危险的，因为初学编程的程序员经常滥用它，并且会打破（在某些情况下）安全屏障。显式内存分配（如通过 `new` 和 `delete` 分配和释放内存）被认为是危险的，因为程序员可能不明智地分配一大块的内存，并且在不需要这块内存的时候忘记释放它，从而引发内存泄漏。尽管这两种特性都具有危险性，但它们使程序员可以直接操作内存，创建高效代码。幸运的是，C++ 不赞成这种理论。

第二种理论（C++ 坚持这种理论）就是“程序员是国王”。这意味着程序员控制一切。语言与您是否成为一位糟糕的程序员无关。语言的主要目标是给程序员一个灵活的、谨慎的工作环境。如果您是一位优秀的程序员，您的工作将会表现出您的优秀。如果您是一位糟糕的程序员，也会通过工作表现出来。总之，C++ 给您提供这些功能，然后就由您自己发挥才能。C++ 程序员永远不会“和语言发生冲突”。

显然，大多数程序员都赞成 C++ 的理论。

1.5 细节控制

C++ 不仅给程序员提供控制的能力，它还提供细节控制。例如，考虑自增运算符：“++”。您知道，C++ 定义了它的前缀和后缀版本。前缀版本在获取其值之前增加操作数。后缀版本在获取值之后增加操作数。当执行包含“++”运算符的表达式时，您获得对表达式执行方式的精确控制。另一个细节控制的示例是 `register` 说明符。通过使用 `register` 说明符修改变量声明，通知编译器优化对此变量的访问。这样，您可以控制哪一个变量具有最高的优化优先权。C++ 给予程序员的细粒度控制能力是 C++ 取代汇编语言成为编写系统代码的首选语言的原因之一。

1.6 运算符重载

C++ 最重要的特性之一就是运算符重载，因为它支持类型扩展。类型扩展可以将新数据类型加入并完全整合到 C++ 程序设计环境中。类型扩展是创建在两个特性之上。第一个是类，允许定义新的数据类型。第二个是运算符重载，允许定义与类相关的多种运算符的含义。通过运算符重载和类，可以创建新的数据类型，然后用操作内建类型相同的方式来操作这种类型；通过运算符。

类型扩展功能非常强大，因为它使得 C++ 成为一个开放而不是封闭的系统。例如，假定需要管理三维坐标。可以通过创建名为 `ThreeD` 的新数据类型，然后定义基于这种类型对象的运算符。例如，可以使用运算符“+”来相加两个 `ThreeD` 坐标，或者使用运算符“==”来判断两组坐标是否相等。这样，可以编写操作 `ThreeD` 的代码，就像操作任何内建类型那样，如下所示：

```
ThreeD a(0, 0, 0), b(1, 2, 3), c(5, 6, 7);
```



```
a = b + c;  
// ...  
if(a == c) // ...
```

如果没有运算符重载,那么对 ThreeD 对象的操作将不得不通过调用函数来处理,如 addThreeD()或者 isEqualThreeD(),这是一种可读性差的方法。

1.7 一种简洁精练的对象模型

C++的对象模型是简洁的杰作。在 C++的 ISO 标准中,对于对象模型的描述还不到一页(准确地讲,是 6 段)。在如此少的段落中,这个标准解释了对对象的本质、对象的生存期以及多态性。例如,这个标准给出了对象的定义,“对象是一个存储区域”。正是这种基础的定义使得 C++的对象模型如此的出众。

当然,在 C++标准中花费了很多的页面来描述为了支持对象所必须的语法和语义,包括对象的创建、销毁和继承等。然而,这是因为 C++给予了对对象的深入控制,而不是因为对象模型的怪异或者不一致。更重要的是,由于优雅的设计,C++对象模型也是被 Java 和 C#语言采用的模型。

1.8 C++发展史

由 Dennis Ritchie 于 20 世纪 70 年代创建的 C 语言标志着程序设计的根本性转变的开始。尽管某些早期的语言,特别是 Pascal,已经获得了巨大的成功,然而 C 语言创建了影响计算机语言产生的范例。C 语言标志着程序设计新时代的开始。

在 C 语言创建之后不久,出现了新的概念:面向对象的程序设计(OOP)。尽管我们现在认为 OOP 的出现是理所当然的,但是在发明它的那个时代,这确实向前迈出了重要的一步。面向对象的理念很快吸引了程序员的注意,因为它提供了一种强大的新方法来完成程序设计工作。在那个时候,程序变得越来越大,并且其复杂度也在增加。因此需要采取一些措施来处理这种复杂性,OOP 提供了一种解决方案。OOP 使得复杂的大程序可以划分为功能性的单元(对象)。这样做使得复杂的系统分解为容易管理的部分。随之出现的问题是 C 语言不支持对象。

由 Bjarne Stroustrup 设计的 C++语言建立在 C 语言的基础之上。Stroustrup 向 C 语言中加入了面向对象程序设计需要的新的关键字和语法。通过向流行的 C 语言加入面向对象特性,Stroustrup 使得成千上万的程序员转向 OOP 成为可能。随着 C++语言的创建,程序设计的新纪元完全实现了。用一个权威人士的话来说,Stroustrup 创建了世界上功能最强大的计算机语言,并且指明了未来语言发展的方向。

尽管 C++语言的发展刚刚开始,但它已经导致了两种重要语言的出现:Java 和 C#。除了稍有区别之外,Java 和 C#的语法、对象模型以及全部的“外观和感受”都非常类似于 C++。另外,Java 和 C#的库的设计中也有 C++的影子,Java 和 C#的 Collection 架构直接由 STL 派生而来。C++的奠基设计对于整个程序设计影响巨大。

C++给程序员提供的强大功能是 C++如此重要的原因。它广泛的影响是它一直成为全世界程序员的卓越语言的原因。

第 2 章 简单的 C++ 垃圾回收器

回顾整个计算的历史，关于管理动态分配内存使用的最优方法一直存在争议。动态分配的内存是在运行期间从堆中获得的内存，堆是供程序使用的自由存储区域。堆通常也称为自由存储区或者动态内存。动态分配非常重要，因为它使得程序在执行期间可以获取、使用、释放，然后重用内存。因为几乎所有实际的程序都会以某种形式使用动态分配，所以管理动态分配的内存的方法对于程序的结构和性能有着深远的影响。

通常，有两种方法处理动态内存。第一种方法是手工方式，程序员必须显式地释放不再使用的内存，从而使得这块内存可以重用。第二种方法依赖于一种自动处理，通常称其为垃圾回收，当某块内存不再需要时会自动被回收器回收。两种方法各有千秋，随着时间的推移，比较好的策略是交替使用这两种方法。

C++使用手工的方法管理动态内存。Java 和 C#使用垃圾回收机制。Java 和 C#是比较新的语言，当前计算机语言设计的倾向是使用垃圾回收器。然而，这并不意味着 C++的程序员被放置在“历史的对立面”。因为 C++具有内建的强大功能，为 C++建立一个垃圾回收器是可能的，甚至是很容易的。因此，C++程序员拥有两者的优点：既可以手工管理动态内存的分配，也可以在需要时使用自动的垃圾回收器。

本章为 C++开发一个完整的垃圾回收子系统。在开始时，很重要的一点是要理解这个垃圾回收器并没有取代 C++内建的动态分配方法。而只是对它进行了补充。因此，在同一个程序中可以使用手工的管理和垃圾回收系统。

本书选用垃圾回收器作为第一个示例，是因为不仅代码本身是有用的(并且是迷人的)，而且它还显示了 C++不可超越的强大功能。通过使用模板类、运算符重载以及 C++的继承能力来处理计算机操纵的底层元素，如内存地址，可以透明地向 C++中加入核心特征。对于大多数的其他语言来说，改变处理动态分配的方式需要改变编译器本身。然而，由于 C++给予程序员的空前强大的功能，这个任务可以在源代码层次上完成。

这个垃圾回收器还显示了如何将一个新的类型定义并整合到 C++的程序设计环境中。这种类型可扩展性(type extensibility)是 C++的关键部分，经常会被忽略。最后，这个垃圾回收器证明了 C++“与机器联系紧密”的能力，因为它操纵并管理了指针。不同于其他阻止对底层细节访问的语言，C++允许程序员在需要的时候充分接近硬件。

2.1 两种内存管理方法的比较

在为 C++开发垃圾回收器之前，比较垃圾回收和内建于 C++中的手工方法是有好处的。通常，在 C++中使用动态内存需要两个步骤。首先，通过 new 从堆中分配内存。然后在不需要这块内存的时候，使用 delete 释放它。因此，每一次动态分配都要遵循下面的顺序：

```
p = new some_object;
```

```
//...
```

```
delete p;
```

通常，每一次使用 `new` 分配内存后，都必须有匹配的 `delete` 操作来释放内存。如果不使用 `delete`，内存就不会被释放，即使您的程序已经不再需要这块内存。

垃圾回收在一个关键方式上不同于手工方法：它自动释放不再需要的内存。因此，通过使用垃圾回收，动态分配只需要一步操作。例如，在 Java 和 C# 中，使用 `new` 分配需要的内存，但是在程序中绝对不需要显式地释放它。相反，垃圾回收器会定期运行，查找不再有其他对象指向的内存块。当没有其他对象指向一个动态内存块时，就意味着程序的元素不再使用这块内存。当找到一块不再使用的内存时，垃圾回收器就会释放它。因此，在一个垃圾回收系统中，没有 `delete` 运算符，也不需要。

乍看上去，垃圾回收的内在简单性使得它成为管理动态内存显而易见的选择。事实上，人们可能会有疑问，究竟为什么要使用手工方法，特别是对于 C++ 这样一种成熟的语言。然而，在动态分配的时候，第一感觉是具有欺骗性的，因为这两种方法都涉及到一组权衡。哪一种方法更好是由应用程序决定的。下面部分描述了一些涉及到的问题。

2.1.1 手工内存管理的优缺点

手工管理动态内存的主要优点是效率。由于没有使用垃圾回收器，从而不需要花费时间来跟踪活动的对象或者周期性地查找不再使用的内存。而是当程序员知道分配的对象不再需要这块内存的时候，他可以显式地释放它，而不需要多余的开销。由于没有垃圾回收相关的开销，手工方法可以编写更加高效的代码。这就是 C++ 需要支持手工内存管理的原因之一：它能够建立高性能代码。

手工管理的另一个优点是控制。尽管要求程序员同时处理内存的分配和释放是一个负担，但这样做的好处是程序员获得了对这个过程两个方面的完全控制。您精确地知道分配内存的时刻，也精确地知道释放它的时刻。另外，当通过 `delete` 释放一个对象的时候，其析构函数在这个时刻执行，而不是像垃圾回收那样在后面的某个时候执行。因此，通过手工方法，可以精确地控制指定对象销毁的时刻。

尽管手工内存管理的效率高，但是它也容易导致相当恼人的一类错误：内存泄漏。由于必须手工释放内存，可能(甚至很容易)忘记这样做。忘记释放不再使用的内存意味着这块内存仍然被分配，即使不再需要它。但在垃圾回收环境中不会发生内存泄漏，因为垃圾回收器确保不再使用的对象最终会被释放。在 Windows 程序设计中，内存泄漏尤其恼人，在这个系统中忘记释放不再使用的内存会逐渐地降低系统性能。

C++ 的手工方法可能涉及到的其他问题包括：过早地释放了仍然在使用的内存，或者不小心将同一块内存释放了两次。这两种错误都会导致严重的问题。而且它们不会立即显示任何征兆，这就使得很难发现这类错误。

2.1.2 垃圾回收的优缺点

实现垃圾回收有多种方法，每一种方法都提供了不同的性能特征。然而，所有的垃圾回收系统都具有一个共同的、与手工方法相对的属性。垃圾回收最主要的优点是简单和安全。在垃

圾回收环境中，可以显式地使用 `new` 分配内存，但是不需要显式地释放内存。相反，不再使用的内存会被自动回收。因此，不可能忘记释放对象或者过早地释放对象。这样做简化了程序设计，并且阻止了有问题的类。另外，不可能意外地两次释放动态分配的内存。因此，垃圾回收为内存管理问题提供了一种易于使用的、不容易犯错的、可靠的解决方案。

遗憾的是，垃圾回收的简单及安全性是有代价的。第一个代价是垃圾回收机制引起的开销。所有垃圾回收的配置都会消耗一些 CPU 资源，因为这种不再使用的内存的回收并不是一个免费过程。当使用手工方法的时候，不会有这样的开销。

第二个代价是在销毁对象时容易失控。使用手工方法时，当对对象执行 `delete` 语句的时候，及时地销毁这个对象(和所调用的它的析构函数)，而垃圾回收没有这种切实而快速的规则。相反，当使用垃圾回收时，直到回收器运行并回收对象的时候，对象才会被销毁，而回收器只有在某个特定时刻才会运行。例如，回收器只有在自由内存的数量低于某个值的时候才会运行。另外，用户并不能总是知道垃圾回收器销毁对象的顺序和时间。在某些情况下，不能准确地知道对象销毁的时间会导致一些问题，因为这意味着程序不能准确地知道何时为动态分配的对象调用析构函数。

对于作为后台任务运行的垃圾回收系统，这种失控可能会引发某种应用程序潜在的更加严重的问题，因为这样做将某种本质上不确定的行为引入到程序中。在后台运行的垃圾回收器实际上在不可预知的某个时刻回收不再使用的内存。例如，回收器通常只有在 CPU 空闲的时候才会运行。由于可能从一个程序的运行转到下一个程序，从一台计算机转到下一台计算机，或者从一个操作系统转到另一个操作系统，因此垃圾回收器在程序中执行的确切位置是不能确定的。对于许多应用程序而言，这并不存在问题，但是对于实时应用程序这可能会引发灾难，因为在实时应用程序中对垃圾回收器不可预知的 CPU 循环的分配会导致事件的丢失。

2.1.3 两种方法都可以使用

正如前面讨论所阐述的那样，手工管理和垃圾回收都强化了一个特性而牺牲了另一个特性。手工方法强化了效率和控制，牺牲了安全性和易用性。垃圾回收强化了简单性及安全性，但是付出了运行性能降低和控制丢失的代价。因此，垃圾回收及手工内存管理本质上是相对的，每一种方法都强化了另一种方法牺牲的特性。这就是没有一种动态内存管理的方法可以适用于所有的程序设计情况的原因。

尽管这两种方法是对立的，但是它们并不互相排斥，它们可以共存。因此对于 C++ 程序员，这两种方法都可以使用，只需为手头的任务选择一种合适的方法。所要做的事情只是为 C++ 建立一个垃圾回收器，这就是本章下面部分的主题。

2.2 在 C++ 中创建垃圾回收器

由于 C++ 是一种功能丰富、强大的语言，因此有许多不同的方法可以建立垃圾回收器。一个明显但受限制的方法就是基于类建立一个垃圾回收器，想要使用垃圾回收的类可以从这个类继承。这样做可以使得一个类接一个类地实现垃圾回收。但遗憾的是，这种解决方案太有限而不能令人满意。

更好的解决方案是建立一个可以被任何动态分配的对象使用的垃圾回收器。为了提供这种

解决方案，垃圾回收器必须满足以下要求：

- (1) 与内建的、C++提供的手工方法共存。
- (2) 不要破坏任何预先存在的代码。更重要的是，不应该对现有的代码造成任何影响。
- (3) 透明地运行，从而利用垃圾回收的分配与不使用这个方案的运行方式相似。
- (4) 类似于 C++ 内建的方法，使用 `new` 来分配内存。
- (5) 对所有的数据类型都有效，包括内建的类型，如 `int` 和 `double` 类型。
- (6) 易于使用。

总之，垃圾回收系统必须以非常接近 C++ 已经使用的机制和语法来动态分配内存，并且不能影响现有的代码。乍看上去，这好像是一个令人生畏的任务，但事实并非如此。

理解这个问题

在建立垃圾回收器的时候，面临的关键问题是如何知道某一块内存不再使用。为了理解这个问题，考虑下面的序列：

```
int *p;
p = new int(99);
p = new int(100);
```

在此动态分配了两个 `int` 对象。第一个对象包含的值为 99，指向这个值的指针存储在 `p` 中。然后，分配了一个值为 100 的整型数据，其地址也存储在 `p` 中，从而改写了第一个地址。在此，`p` (或者其他对象) 不再指向 `int(99)` 的那块内存，从而可以释放这块内存。问题是，垃圾回收器如何知道 `p` 或者其他的对象都不指向 `int(99)` 了呢？

对这个问题稍做变动：

```
int *p, *q;
p = new int(99);
q = p; // now, q points to same memory as p
p = new int(100);
```

在此情况下，`q` 指向了原先为 `p` 分配的那块内存。即使 `p` 指向了不同的内存块，原先指向的内存也不能被释放，因为 `q` 仍然在使用它。问题是：垃圾回收器如何知道这一事实呢？对这个问题的准确回答取决于垃圾回收所使用的算法。

2.3 选择垃圾回收的算法

在为 C++ 实现垃圾回收器之前，有必要确定垃圾回收使用的算法。垃圾回收的内容很多，很多年来一直是学术研究的焦点。因为它提出了一个吸引人的问题，对于这个问题有多种解决方案，并且设计了许多不同的垃圾回收算法。详细地验证每一种方法远远超出了本书讨论的范围。然而，在此有 3 种典型方法：引用计数、标记并清除，以及复制。在确定要选择哪种方法之前，有必要先浏览一下这 3 种算法。

2.3.1 引用计数

在引用计数中，每一块动态分配的内存都与一个引用计数相关。这个计数在每次对内存的引用增加的时候增 1，在取消对内存的引用时减 1。用 C++ 的术语来说，这意味着每次将一个指针指向一块已分配内存的时候，与内存相关的引用计数增 1。当这个指针指向其他位置的时候，引用计数减 1。当引用计数下降为 0 的时候，内存不再被使用，从而可以释放。

引用计数的最大优点是其简单性——易于理解并实现。另外，它的位置不受堆结构的影响，因为引用计数不依赖于对象的物理位置。引用计数增加了每个指针操作的开销，但是回收阶段的开销相对较低。其主要的缺点是循环的引用阻止了其他不再使用的内存的释放。当两个对象互相指向对方的时候(无论是直接的还是间接的)，就会发生循环引用。在此情况下，对象的引用计数永不为 0。为了解决循环引用的问题，设计了一些解决方案，但是这些方案都会增加复杂程度和/或开销。

2.3.2 标记并清除

标记并清除涉及到两个阶段。在第一个阶段，堆中的所有对象都被设置为未标记状态。然后，可以由程序变量直接或者间接访问的所有对象都被标记为“正在使用”。在第二个阶段，扫描所有已分配的内存(也就是说，进行了内存的清除)，会释放所有未标记的元素。

标记并清除有两个主要优点。首先，它很容易处理循环引用。其次，在回收之前，它实际上没有增加运行时开销。它也有两个主要缺点。首先，由于在回收的时候必须扫描整个堆，因此回收垃圾可能会花费较多的时间。因此，对于某些程序，垃圾回收可能会导致程序运行效率低下。其次，尽管标记并清除在概念上很简单，但是要有效地实现它并非易事。

2.3.3 复制

复制算法将自由内存分到两个空间中。一个是活动空间(持有当前的堆)，一个是空闲空间。在垃圾回收期间，活动空间中正在使用的对象被确认，并复制到空闲空间中。然后，两个空间的角色反转，空闲空间变为活动空间，活动空间变为空闲空间。复制提供了复制过程中压缩堆的优点。它的缺点是在某个时刻只允许使用一半的自由内存。

2.3.4 采用哪种算法

三种垃圾回收的经典方法都有各自的优缺点，好像很难做出选择。然而，考虑前面列举出的限制，就会得出明显的选择：引用计数。最重要的是，引用计数可以很容易地应用于现有的 C++ 动态分配系统上。其次，它可以以一种直接的方式来实现，而不会影响现有的代码。第三，它不需要堆的任何特定的组织或者结构，从而不会影响 C++ 提供的标准分配系统。

使用引用计数的一个缺点是很难处理循环引用。这对于许多程序而言，并不是一个问题，因为有意的循环引用并不常用，并且可以避免。(即使我们所说的循环，如循环队列，也不一定要用到循环指针引用)。当然，某些情况下需要使用循环引用。也可能建立了循环引用，而您并不知道，特别是使用第三方库的时候。因此，垃圾回收器必须提供某种方法来适度地处理循环引用。

为了处理循环引用问题，本章开发的垃圾回收器在程序退出的时候，释放任何已经分配的内存。这将确保涉及到循环引用的对象被释放，并且调用它们的析构函数。通常在程序结束的

时候, 不应该再有已分配的对象, 理解这一点很重要。对于涉及到循环引用而不能被释放的对象, 这种机制是显式的。(您可能会试验用其他的方法处理循环引用。这是一个有趣的挑战)。

2.3.5 实现垃圾回收器

为了实现引用计数的垃圾回收器, 必须有某种方法来跟踪指向每块动态分配内存的指针的数量。问题在于, C++没有内建的机制来确保一个对象知道其他的对象何时指向它。幸运的是, 在此有一个解决方案: 可以建立一个新的支持垃圾回收的指针类型。这就是本章的垃圾回收器采用的方法。

为了支持垃圾回收, 新的指针类型必须做三件事情。第一, 它必须为使用中的动态分配的对象维护一个引用计数的链表。第二, 它必须跟踪所有的指针运算符, 每次某个指针指向一个对象时, 都要使这个对象的引用计数增 1; 每次某个指针重新指向其他对象的时候, 都要使这个对象的引用计数减 1。第三, 它必须回收那些引用计数降为 0 的对象。除了支持垃圾回收之外, 这个新指针类型与普通的指针看起来一样。例如, 它支持所有的指针运算, 如*和->运算。

垃圾回收指针类型的建立不仅以一种简便的方法实现垃圾回收器, 而且还满足不影响原始的 C++动态分配系统的限制。当需要垃圾回收的时候, 使用支持垃圾回收的指针。当不需要垃圾回收的时候, 使用普通的 C++指针。因此, 这两种类型的指针在同一程序内都可以使用。

2.3.6 是否使用多线程

在设计 C++的垃圾回收器时, 另一个考虑是应该使用单线程还是多线程。也就是说, 是否应该把垃圾回收器设计为一个后台进程, 在它自己的线程内运行, 并且在 CPU 时间允许时回收垃圾。或者, 这个垃圾回收器在使用它的进程的相同线程中运行, 当满足某种程序条件的时候回收垃圾。两种方法各有优缺点。

建立多线程垃圾回收器的主要优点是效率。垃圾可以在 CPU 空闲时被回收。其缺点是, C++没有提供内建的多线程支持。这意味着为了支持多任务, 任何多线程方法都依赖于操作系统是否支持多任务。这使得代码不可移植。

使用单线程垃圾回收器的主要优点是代码可以移植。在不支持多线程或者支持多线程的代价很高的情况下使用。主要缺点是当垃圾回收发生时, 程序的其他部分会停止运行。

在本章使用了单线程的方法, 因为在所有的 C++环境中, 它都可以使用。因此, 本书所有的读者都可以使用它。然而, 对于那些想要使用多线程解决方案的用户, 第 3 章给出了一个示例, 来处理在 Windows 环境下成功地实现多线程 C++程序所需要的技术。

2.3.7 何时回收垃圾

在实现垃圾回收器之前, 需要回答的最后一个问题是: 什么时候开始垃圾回收? 对于多线程的垃圾回收器这不成问题, 因为它可以作为后台任务连续运行, 并且在 CPU 空闲的时候回收垃圾。然而对于单线程的垃圾回收器, 如本章开发的这个回收器, 为了回收垃圾, 必须停止运行程序的其他部分。

实际上, 只有在有足够的理由(如内存持续降低)的时候, 才会进行垃圾回收。有两个原因使得这样做有意义。首先, 通过某种垃圾回收算法, 如标记并清除, 如果不实际执行回收, 就没有办法知道不再使用的某块内存。(也就是说, 如果不实际回收垃圾, 某些时候没有办法知道它的存在), 其次, 回收垃圾是一个耗时的过程, 在不需要的时候不应该执行它。

然而，在开始垃圾回收之前等待内存持续降低不适合本章的目的，因为这样做使得几乎不可能演示回收器。为此，本章开发的垃圾回收器将更加频繁地回收垃圾，这样可以比较容易地观察到它的活动。当给回收器编码完成之后，无论什么时候指针超出了作用域，都会回收垃圾。当然，可以很容易地改变这种行为以适应您的应用程序。

最后一点要注意的是：在使用基于引用计数的垃圾回收时，只要不再使用的内存的引用计数降为 0，技术上就可以回收它，而不需要使用一个独立的垃圾回收阶段。然而，这种方法对每个指针操作都加入了额外的开销。本章所使用的方法是，在指向某块内存的指针重定向之后，简单地减小这块内存的引用计数。这样做降低了与指针运算相关的运行时开销，通常人们总是希望它越快越好。

2.3.8 关于 auto_ptr

许多读者都知道，C++定义了一个称为 auto_ptr 的库类。由于 auto_ptr 在超出作用域时，它指向的内存会自动释放，您可能会认为在开发垃圾回收器时，它会有用，或许它建立了这个垃圾回收器的基础。然而事实上并非如此。auto_ptr 类是为 ISO C++标准称为“严格所有权”的概念设计的，其中 auto_ptr “拥有”它指向的对象。这种所有权可以转换到其他的 auto_ptr，但是在任何情况下，总会有某个 auto_ptr 拥有这个对象，直到它被销毁。另外，auto_ptr 只有在初始化时，才能使用对象的地址对其赋值。在此之后，您不能够改变 auto_ptr 指向的内存，除非将一个 auto_ptr 赋值给另一个 auto_ptr。由于 auto_ptr 具有“严格所有权”的特征，在建立垃圾回收器时，它没有实际的用处，从而也不会在这本书中的垃圾回收器中使用。

2.4 一个简单的 C++垃圾回收器

在此给出了一个完整的垃圾回收器。如前所述，这个垃圾回收器通过建立一个新的指针类型来运行，这个新的指针类型基于引用计数，为垃圾回收提供内在的支持。这个垃圾回收器是单线程的，这意味着它具有很好的移植性，并且不依赖于(或者对其做一些假定)执行环境。这些代码存储在文件 gc.h 中。

当您阅读代码时，需要注意两件事情。首先，大多数成员函数都相当简短，为了提高效率，都在各自的类中定义。回想一下，在类中定义的函数都自动成为内联函数，这样做消除了函数调用的开销。仅有少数函数比较长，因而需要在类的外部定义它们。

其次，注意文件头的注释。如果您想要观察垃圾回收器是如何工作的，只需要简单地通过定义 DISPLAY 宏来开启显示选项。对于普通的使用，不需要定义 DISPLAY。

```
// A single-threaded garbage collector.

#include <iostream>
#include <list>
#include <typeinfo>
#include <cstdlib>

using namespace std;

// To watch the action of the garbage collector, define DISPLAY.
```



```

// #define DISPLAY

// Exception thrown when an attempt is made to
// use an Iter that exceeds the range of the
// underlying object.
//
class OutOfRangeExc {
    // Add functionality if needed by your application.
};

// An iterator-like class for cycling through arrays
// that are pointed to by GCPtrs. Iter pointers
// ** do not ** participate in or affect garbage
// collection. Thus, an Iter pointing to
// some object does not prevent that object
// from being recycled.
//
template <class T> class Iter {
    T *ptr; // current pointer value
    T *end; // points to element one past end
    T *begin; // points to start of allocated array
    unsigned length; // length of sequence
public:

    Iter() {
        ptr = end = begin = NULL;
        length = 0;
    }

    Iter(T *p, T *first, T *last) {
        ptr = p;
        end = last;
        begin = first;
        length = last - first;
    }

    // Return length of sequence to which this
    // Iter points.
    unsigned size() { return length; }

    // Return value pointed to by ptr.
    // Do not allow out-of-bounds access.
    T &operator*() {
        if( (ptr >= end) || (ptr < begin) )
            throw OutOfRangeExc();
        return *ptr;
    }

    // Return address contained in ptr.
    // Do not allow out-of-bounds access.
    T *operator->() {
        if( (ptr >= end) || (ptr < begin) )

```

```

        throw OutOfRangeExc();
    return ptr;
}

// Prefix ++.
Iter operator++() {
    ptr++;
    return *this;
}

// Prefix --.
Iter operator--() {
    ptr--;
    return *this;
}

// Postfix ++.
Iter operator++(int notused) {
    T *tmp = ptr;

    ptr++;
    return Iter<T>(tmp, begin, end);
}

// Postfix --.
Iter operator--(int notused) {
    T *tmp = ptr;

    ptr--;
    return Iter<T>(tmp, begin, end);
}

// Return a reference to the object at the
// specified index. Do not allow out-of-bounds
// access.
T &operator[](int i) {
    if( (i < 0) || (i >= (end-begin)) )
        throw OutOfRangeExc();
    return ptr[i];
}

// Define the relational operators.
bool operator==(Iter op2) {
    return ptr == op2.ptr;
}

bool operator!=(Iter op2) {
    return ptr != op2.ptr;
}

bool operator<(Iter op2) {
    return ptr < op2.ptr;
}

```

```

    }

    bool operator<=(Iter op2) {
        return ptr <= op2.ptr;
    }

    bool operator>(Iter op2) {
        return ptr > op2.ptr;
    }

    bool operator>=(Iter op2) {
        return ptr >= op2.ptr;
    }

    // Subtract an integer from an Iter.
    Iter operator-(int n) {
        ptr -= n;
        return *this;
    }

    // Add an integer to an Iter.
    Iter operator+(int n) {
        ptr += n;
        return *this;
    }

    // Return number of elements between two Iters.
    int operator-(Iter<T> &itr2) {
        return ptr - itr2.ptr;
    }

};

// This class defines an element that is stored
// in the garbage collection information list.
//
template <class T> class GCInfo {
public:
    unsigned refcount; // current reference count

    T *memPtr; // pointer to allocated memory

    /* isArray is true if memPtr points
       to an allocated array. It is false
       otherwise. */
    bool isArray; // true if pointing to array

    /* If memPtr is pointing to an allocated
       array, then arraySize contains its size */
    unsigned arraySize; // size of array

```

```

// Here, mPtr points to the allocated memory.
// If this is an array, then size specifies
// the size of the array.
GCInfo(T *mPtr, unsigned size=0) {
    refcount = 1;
    memPtr = mPtr;
    if(size != 0)
        isArray = true;
    else
        isArray = false;

    arraySize = size;
}
};

// Overloading operator== allows GCInfos to be compared.
// This is needed by the STL list class.
template <class T> bool operator==(const GCInfo<T> &ob1,
                                   const GCInfo<T> &ob2) {
    return (ob1.memPtr == ob2.memPtr);
}

// GCPtr implements a pointer type that uses
// garbage collection to release unused memory.
// A GCPtr must only be used to point to memory
// that was dynamically allocated using new.
// When used to refer to an allocated array,
// specify the array size.
//
template <class T, int size=0> class GCPtr {

    // gclist maintains the garbage collection list.
    static list<GCInfo<T> > gclist;

    // addr points to the allocated memory to which
    // this GCPtr pointer currently points.
    T *addr;

    /* isArray is true if this GCPtr points
       to an allocated array. It is false
       otherwise. */
    bool isArray; // true if pointing to array

    // If this GCPtr is pointing to an allocated
    // array, then arraySize contains its size.
    unsigned arraySize; // size of the array
    static bool first; // true when first GCPtr is created

    // Return an iterator to pointer info in gclist.
    typename list<GCInfo<T> >::iterator findPtrInfo(T *ptr);
}

```

```

public:

    // Define an iterator type for GCPtr<T>.
    typedef Iter<T> GCiterator;

    // Construct both initialized and uninitialized objects.
    GCPtr(T *t=NULL) {

        // Register shutdown() as an exit function.
        if(first) atexit(shutdown);
        first = false;

        list<GCInfo<T> >::iterator p;

        p = findPtrInfo(t);

        // If t is already in gclist, then
        // increment its reference count.
        // Otherwise, add it to the list.
        if(p != gclist.end())
            p->refcount++; // increment ref count
        else {
            // Create and store this entry.
            GCInfo<T> gcObj(t, size);
            gclist.push_front(gcObj);
        }

        addr = t;
        arraySize = size;
        if(size > 0) isArray = true;
        else isArray = false;
#ifdef DISPLAY
        cout << "Constructing GCPtr. ";
        if(isArray)
            cout << " Size is " << arraySize << endl;
        else
            cout << endl;
#endif
    }

    // Copy constructor.
    GCPtr(const GCPtr &ob) {
        list<GCInfo<T> >::iterator p;

        p = findPtrInfo(ob.addr);
        p->refcount++; // increment ref count

        addr = ob.addr;
        arraySize = ob.arraySize;
        if(arraySize > 0) isArray = true;
        else isArray = false;
#ifdef DISPLAY

```

```

        cout << "Constructing copy.";
        if(isArray)
            cout << " Size is " << arraySize << endl;
        else
            cout << endl;
    #endif
}

// Destructor for GCPtr.
~GCPtr();

// Collect garbage. Returns true if at least
// one object was freed.
static bool collect();

// Overload assignment of pointer to GCPtr.
T *operator=(T *t);

// Overload assignment of GCPtr to GCPtr.
GCPtr &operator=(GCPtr &rv);

// Return a reference to the object pointed
// to by this GCPtr.
T &operator*() {
    return *addr;
}

// Return the address being pointed to.
T *operator->() { return addr; }

// Return a reference to the object at the
// index specified by i.
T &operator[](int i) {
    return addr[i];
}

// Conversion function to T *.
operator T *() { return addr; }

// Return an Iter to the start of the allocated memory.
Iter<T> begin() {
    int size;

    if(isArray) size = arraySize;
    else size = 1;

    return Iter<T>(addr, addr, addr + size);
}

// Return an Iter to one past the end of an allocated array.
Iter<T> end() {
    int size;

```

```

    if(isArray) size = arraySize;
    else size = 1;

    return Iter<T>(addr + size, addr, addr + size);
}

// Return the size of gclist for this type
// of GCPtr.
static int gclistSize() { return gclist.size(); }

// A utility function that displays gclist.
static void showlist();

// Clear gclist when program exits.
static void shutdown();
};

// Creates storage for the static variables
template <class T, int size>
    list<GCInfo<T> > GCPtr<T, size>::gclist;

template <class T, int size>
    bool GCPtr<T, size>::first = true;

// Destructor for GCPtr.
template <class T, int size>
GCPtr<T, size>::~~GCPtr() {
    list<GCInfo<T> >::iterator p;

    p = findPtrInfo(addr);
    if(p->refcount) p->refcount--; // decrement ref count

#ifdef DISPLAY
    cout << "GCPtr going out of scope.\n";
#endif

    // Collect garbage when a pointer goes out of scope.
    collect();

    // For real use, you might want to collect
    // unused memory less frequently, such as after
    // gclist has reached a certain size, after a
    // certain number of GCPtrs have gone out of scope,
    // or when memory is low.
}

// Collect garbage. Returns true if at least
// one object was freed.
template <class T, int size>
bool GCPtr<T, size>::collect() {
    bool memfreed = false;

```

```

#ifdef DISPLAY
    cout << "Before garbage collection for ";
    showlist();
#endif

list<GCInfo<T> >::iterator p;
do {

    // Scan gclist looking for unreferenced pointers.
    for(p = gclist.begin(); p != gclist.end(); p++) {
        // If in-use, skip.
        if(p->refcount > 0) continue;

        memfreed = true;

        // Remove unused entry from gclist.
        gclist.remove(*p);
        // Free memory unless the GCPtr is null.
        if(p->memPtr) {
            if(p->isArray) {
                #ifdef DISPLAY
                    cout << "Deleting array of size "
                        << p->arraySize << endl;
                #endif
                delete[] p->memPtr; // delete array
            }
            else {
                #ifdef DISPLAY
                    cout << "Deleting: "
                        << *(T *) p->memPtr << "\n";
                #endif
                delete p->memPtr; // delete single element
            }
        }

        // Restart the search.
        break;
    }

} while(p != gclist.end());

#ifdef DISPLAY
    cout << "After garbage collection for ";
    showlist();
#endif

return memfreed;
}

// Overload assignment of pointer to GCPtr.
template <class T, int size>

```



```

T * GCPtr<T, size>::operator=(T *t) {
    list<GCInfo<T> >::iterator p;

    // First, decrement the reference count
    // for the memory currently being pointed to.
    p = findPtrInfo(addr);
    p->refcount--;

    // Next, if the new address is already
    // existent in the system, increment its
    // count. Otherwise, create a new entry
    // for gclist.
    p = findPtrInfo(t);
    if(p != gclist.end())
        p->refcount++;
    else {
        // Create and store this entry.
        GCInfo<T> gcObj(t, size);
        gclist.push_front(gcObj);
    }

    addr = t; // store the address.

    return t;
}

// Overload assignment of GCPtr to GCPtr.
template <class T, int size>
GCPtr<T, size> & GCPtr<T, size>::operator=(GCPtr &rv) {
    list<GCInfo<T> >::iterator p;

    // First, decrement the reference count
    // for the memory currently being pointed to.
    p = findPtrInfo(addr);
    p->refcount--;

    // Next, increment the reference count of
    // the new address.
    p = findPtrInfo(rv.addr);
    p->refcount++; // increment ref count

    addr = rv.addr; // store the address.

    return rv;
}

// A utility function that displays gclist.
template <class T, int size>
void GCPtr<T, size>::showlist() {
    list<GCInfo<T> >::iterator p;

    cout << "gclist<" << typeid(T).name() << ", "

```

```

        << size << ">:\n";
    cout << "memPtr refcount value\n";

    if(gclist.begin() == gclist.end()) {
        cout << " -- Empty --\n\n";
        return;
    }

    for(p = gclist.begin(); p != gclist.end(); p++) {
        cout << "[" << (void *)p->memPtr << "]"
            << " " << p->refcount << " ";
        if(p->memPtr) cout << " " << *p->memPtr;
        else cout << " ---";
        cout << endl;
    }
    cout << endl;
}

// Find a pointer in gclist.
template <class T, int size>
typename list<GCInfo<T> >::iterator
GCPtr<T, size>::findPtrInfo(T *ptr) {

    list<GCInfo<T> >::iterator p;

    // Find ptr in gclist.
    for(p = gclist.begin(); p != gclist.end(); p++)
        if(p->memPtr == ptr)
            return p;

    return p;
}

// Clear gclist when program exits.
template <class T, int size>
void GCPtr<T, size>::shutdown() {

    if(gclistSize() == 0) return; // list is empty

    list<GCInfo<T> >::iterator p;

    for(p = gclist.begin(); p != gclist.end(); p++) {
        // Set all reference counts to zero
        p->refcount = 0;
    }

#ifdef DISPLAY
    cout << "Before collecting for shutdown() for "
        << typeid(T).name() << "\n";
#endif

    collect();
}

```

```

#ifdef DISPLAY
    cout << "After collecting for shutdown() for "
        << typeid(T).name() << "\n";
#endif
}

```

垃圾回收器类概述

垃圾回收器使用了 4 个类：GCPtr、GCInfo、Iter 以及 OutOfRangeExc。在详细分析这些代码之前，有必要先理解每一个类所扮演的角色。

1. GCPtr

垃圾回收器的核心是 GCPtr 类，它实现了垃圾回收指针。GCPtr 维护了一个链表，这个链表与 GCPtr 分配的每一块内存的引用计数相关。下面是它的运行方式。每次 GCPtr 指向一块内存时，这块内存的引用计数增 1。如果 GCPtr 在分配之前指向不同的内存块，那么这块内存的引用计数减 1。向某一块内存加入指针增加了这块内存的引用计数，移除指针减少了这块内存的引用计数。每一次 GCPtr 超出作用域，当前它所指向的内存的引用计数都会减小。当引用计数降为 0 时，这一块内存就被释放了。

GCPtr 是一个重载了“*”、“->”指针运算符以及数组索引运算符[]的模板类。因此，GCPtr 建立了一个新的指针类型，并且将它整合到 C++ 的程序设计环境中。从而使得能够以类似于使用普通 C++ 指针的方式来使用 GCPtr。然而在本章，为了清楚起见，GCPtr 没有重载++、-- 以及其他为指针定义的算术运算符。因此，除了通过赋值之外，您不能改变 GCPtr 对象所指向的地址。这好像是一个很大的限制，但事实上并非如此，因为 Iter 类提供了这些操作。

为了说明问题，无论何时 GCPtr 对象超出作用域，垃圾回收器都会运行。这时会扫描垃圾回收链表，所有引用计数为 0 的内存都会被释放，即使与超出作用域的 GCPtr 无关的内存也是如此。如果您在前面就需要回收内存，就可以显式地要求垃圾回收。

2. GCInfo

如前所述，GCPtr 维护了一个将引用计数与已分配内存链接起来的链表。这个链表中的每一项都封装在一个 GCInfo 类型的对象之中。GCInfo 在它的 refcount 字段中存储了引用计数，在 memPtr 字段中保存了这块内存的指针。因此，GCInfo 对象将引用计数与已分配内存绑定在一起。

GCInfo 定义了两个其他字段：isArray 和 arraySize。如果 memPtr 指向了一个已分配的数组，那么 isArray 成员就为 true，这个数组的长度就存储在它的 arraySize 字段中。

3. Iter 类

前面已经说过，GCPtr 对象允许使用普通的指针运算符“*”和“->”来访问它所指向的内存，但是它不支持指针运算。为了处理需要执行这些指针运算的情况，可以使用 Iter 类型的对象。Iter 是一个功能上类似于 STL 迭代器的模板类，它定义了所有的指针操作，包括指针运算。Iter 的主要用处是遍历动态分配的数组元素。它还提供边界检查。可以通过调用 GCPtr 的 begin() 或者 end() 来获取 Iter 对象，其运行方式与 STL 中的迭代器非常类似。

尽管 Iter 与 STL 中的迭代器 iterator 类型非常相似，但是它们并不相同，理解这一点很重要，它们不能互相替代使用。

4. OutOfRangeExc

如果 Iter 试图访问已分配内存范围之外的内存，则会抛出 OutOfRangeExc 异常。本章中 OutOfRangeExc 不包含成员。它只是一个可以抛出的类型。然而，如果您的应用程序需要，就可以随意向这个类中添加其他功能。

2.5 详细讨论 GCPtr

GCPtr 是垃圾回收器的核心。它实现了一个新的指针类型来保存堆中已分配对象的引用计数。它还提供了垃圾回收的功能来回收不再使用的内存。

GCPtr 是一个模板类，其声明如下：

```
template <class T, int size=0> class GCPtr {
```

GCPtr 要求您指定将被指向的数据的类型，这个类型将取代通用类型 T。如果被分配的是一个数组，就必须在 size 参数中指定数组的大小。否则，size 默认为 0，这说明它指向一个单独的对象。在此给出两个示例。

```
GCPtr<int> p; // declare a pointer to a single integer
GCPtr<int, 5> ap; // declare a pointer to an array of 5 integers
```

在此，p 可以指向一个 int 类型的对象，ap 可以指向大小为 5 的 int 数组。

注意在前面的示例中，在指定 GCPtr 对象名称时，没有使用 “*” 运算符。也就是说，不需要使用这样的语句来建立一个指向 int 的 GCPtr：

```
GCPtr<int> *p; // this creates a pointer to a GCPtr<int> object
```

这个声明建立了一个名为 p 的指向一个 GCPtr<int> 对象的普通 C++ 指针，而不是建立了一个指向 int 的 GCPtr 对象。记住，GCPtr 本身就定义了一个指针类型。

在指定 GCPtr 的类型参数的时候必须谨慎。这个参数指定了 GCPtr 对象可以指向的对象类型。因此，如果您使用了这样的声明：

```
GCPtr<int *> p; // this creates a GCPtr to pointers to ints
```

那就创建了一个指向 int* 指针的 GCPtr 对象，而不是指向 int 的 GCPtr 对象。

由于其重要性，在接下来的部分，详细地分析每个 GCPtr 的成员。

2.5.1 GCPtr 的数据成员

GCPtr 声明了如下的数据成员：

```
// gclist maintains the garbage collection list.
static list<GCInfo<T> > gclist;

// addr points to the allocated memory to which
```

```

// this GCPtr pointer currently points.
T *addr;

/* isArray is true if this GCPtr points
   to an allocated array. It is false
   otherwise. */
bool isArray; // true if pointing to array

// If this GCPtr is pointing to an allocated
// array, then arraySize contains its size.
unsigned arraySize; // size of the array

static bool first; // true when first GCPtr is created

```

`gclist` 字段包含了 `GCInfo` 对象的链表。(回顾一下, `GCInfo` 链接了引用计数与已分配的内存块。)垃圾回收器使用这个链表来判断何时不再使用已分配内存。注意 `gclist` 是 `GCPtr` 的静态成员。这意味着对于每个特定的指针类型, 只会有一个 `gclist`。例如, 所有的 `GCPtr<int>` 类型的指针共享一个链表, 所有的 `GCPtr<double>` 类型的指针共享另一个链表。`gclist` 是 STL `list` 类的一个实例。使用 STL 极大地简化了 `GCPtr` 的代码, 因为在此不需要创建自己的链表处理函数集。

`GCPtr` 在 `addr` 中存储了它所指向的内存的地址。如果 `addr` 指向一个已分配的数组, 那么 `isArray` 为 `true`, 数组的长度存储在 `arraySize` 中。

`first` 字段是一个静态变量, 其初始值设置为 `true`。`GCPtr` 构造函数使用这个标记来了解何时建立了第一个 `GCPtr` 对象。在第一个 `GCPtr` 对象建立之后, `first` 被设置为 `false`。它用来注册一个终止函数, 这个函数在程序终止时调用, 从而关闭垃圾回收器。

2.5.2 函数 `findPtrInfo()`

`GCPtr` 声明了一个私有函数: `findPtrInfo()`。这个函数在 `gclist` 中查找指定的地址, 并且返回该项的一个迭代器。如果没有找到这个地址, 就返回这个 `gclist` 结尾的迭代器。这个函数由 `GCPtr` 在内部使用, 来更新 `gclist` 内的对象的引用计数。其实现如下:

```

// Find a pointer in gclist.
template <class T, int size>
typename list<GCInfo<T> >::iterator
GCPtr<T, size>::findPtrInfo(T *ptr) {

    list<GCInfo<T> >::iterator p;

    // Find ptr in gclist.
    for(p = gclist.begin(); p != gclist.end(); p++)
        if(p->memPtr == ptr)
            return p;

    return p;
}

```

2.5.3 GCIterator typedef

GCPtr 的公有部分以 Iter<T>的 GCIterator typedef 开始。这个 typedef 与 GCPtr 的每个实例绑定，从而不必在每一次 Iter 需要指定 GCPtr 的版本时指定类型参数。这样做简化了迭代器的声明。例如，为了获得由特定 GCPtr 指向的内存的迭代器，可以使用下面的语句：

```
GCPtr<int>::GCIterator itr;
```

2.5.4 GCPtr 的构造函数

GCPtr 的构造函数如下所示：

```
// Construct both initialized and uninitialized objects.
GCPtr(T *t=NULL) {

    // Register shutdown() as an exit function.
    if(first) atexit(shutdown);
    first = false;
    list<GCInfo<T> >::iterator p;

    p = findPtrInfo(t);

    // If t is already in gclist, then
    // increment its reference count.
    // Otherwise, add it to the list.
    if(p != gclist.end())
        p->refcount++; // increment ref count
    else {
        // Create and store this entry.
        GCInfo<T> gcObj(t, size);
        gclist.push_front(gcObj);
    }

    addr = t;
    arraySize = size;
    if(size > 0) isArray = true;
    else isArray = false;
#ifdef DISPLAY
    cout << "Constructing GCPtr. ";
    if(isArray)
        cout << " Size is " << arraySize << endl;
    else
        cout << endl;
#endif
}
```

GCPtr() 允许创建已初始化的实例和未初始化的实例。如果声明了已初始化的实例，那么 GCPtr 将指向的那块内存就会传递给 t。否则，t 就会是 null。让我们分析检查一下 GCPtr() 的操作。

首先，如果 first 为 true，就意味着将要创建第一个 GCPtr 对象。在此情况下，通过调用 atexit()，将 shutdown() 注册为终止函数。atexit() 函数是 C++ 标准函数库的一部分，它用来注册在程序结束时调用的函数。此时，shutdown() 释放任何由于循环引用而没有释放的内存。

然后对 `gclist` 进行搜索，查找任何与 `t` 中的地址匹配的先前存在的条目。如果能够找到，就会增加它的引用计数。如果没有先前存在的条目与 `t` 匹配，就创建包含这个地址的新 `GCInfo` 对象，并且将这个对象加入到 `gclist` 中。

随后 `GCPtr()` 将 `addr` 设置为 `t` 指定的地址，并且正确地设置 `isArray` 和 `arraySize` 的值。记住，如果您分配了一个数组，那么当声明指向它的 `GCPtr` 指针时，必须显式地指定这个数组的大小。如果不这么做的话，这块内存就不会被正确地释放，在类对象数组的情况下，析构函数不会被正确地调用。

2.5.5 GCPtr 的析构函数

`GCPtr` 的析构函数如下：

```
// Destructor for GCPtr.
template <class T, int size>
GCPtr<T, size>::~~GCPtr() {
    list<GCInfo<T> >::iterator p;

    p = findPtrInfo(addr);
    if(p->refcount) p->refcount--; // decrement ref count

#ifdef DISPLAY
    cout << "GCPtr going out of scope.\n";
#endif

    // Collect garbage when a pointer goes out of scope.
    collect();

    // For real use, you might want to collect
    // unused memory less frequently, such as after
    // gclist has reached a certain size, after a
    // certain number of GCPtrs have gone out of scope,
    // or when memory is low.
}
```

每次 `GCPtr` 超出作用域时，垃圾回收都会运行。这是通过 `~GCPtr()` 完成的。首先搜索 `gclist`，查找与将被销毁的 `GCPtr` 所指的地址对应的条目。一旦发现，减少其引用计数。随后，`~GCPtr()` 调用 `collect()` 来释放不再使用的内存(也就是其引用计数为 0 的内存)。

正如 `~GCPtr()` 结尾的注释所讲的那样，对于实际的应用程序，垃圾回收的频率少于 `GCPtr` 超出作用域的频率可能会更好。较小频率的回收通常效率更高。如前所述，每次 `GCPtr` 销毁的时候进行回收对于演示垃圾回收器是有用的，因为这样做清楚地说明了垃圾回收器的运行方式。

2.5.6 回收垃圾函数 `collect()`

使用 `collect()` 函数进行垃圾回收。其代码如下：

```
// Collect garbage. Returns true if at least
// one object was freed.
template <class T, int size>
```

```

bool GCPtr<T, size>::collect() {
    bool memfreed = false;

#ifdef DISPLAY
    cout << "Before garbage collection for ";
    showlist();
#endif

    list<GCInfo<T> >::iterator p;
    do {

        // Scan gclist looking for unreferenced pointers.
        for(p = gclist.begin(); p != gclist.end(); p++) {
            // If in-use, skip.
            if(p->refcount > 0) continue;

            memfreed = true;

            // Remove unused entry from gclist.
            gclist.remove(*p);

            // Free memory unless the GCPtr is null.
            if(p->memPtr) {
                if(p->isArray) {
#ifdef DISPLAY
                    cout << "Deleting array of size "
                        << p->arraySize << endl;
#endif
                    delete[] p->memPtr; // delete array
                }
                else {
#ifdef DISPLAY
                    cout << "Deleting: "
                        << *(T *) p->memPtr << "\n";
#endif
                    delete p->memPtr; // delete single element
                }
            }

            // Restart the search.
            break;
        }

    } while(p != gclist.end());

#ifdef DISPLAY
    cout << "After garbage collection for ";
    showlist();
#endif
    return memfreed;
}

```


`collect()`函数通过扫描 `gclist` 的内容，查找是否有 `refcount` 为 0 的条目。当找到这种条目时，通过调用 `remove()`函数删除它，`remove()`函数是 STL `list` 类的一个成员。然后释放与这个条目相关的内存。

回忆一下，在 C++ 中，单个对象通过 `delete` 释放，而对象数组通过 `delete[]` 释放。因此，条目的 `isArray` 字段的值决定了是使用 `delete` 还是 `delete[]` 来释放内存。这就是您必须为任何指向数组的 `GCPtr` 指定已分配数组的大小的原因之一：这样做将会使得 `isArray` 为 `true`。如果没有正确地设置 `isArray`，就不可能正确地释放已分配的内存。

尽管垃圾回收的主要目的是为了自动回收不再使用的内存，但是在需要的时候也可以采用手工控制的方法。用户代码可以直接调用 `collect()`函数进行垃圾回收。注意这个函数声明为 `GCPtr` 中的静态函数，这意味着不引用任何对象就可以调用它。

例如：

```
GCPtr<int>::collect(); // collect all unused int pointers
```

这会导致 `gclist<int>` 被回收。因为对于每个不同类型的指针，都会有不同的 `gclist`，所以您需要为每个想要回收的链表调用 `collect()`。坦白地讲，如果想要更加明确地管理动态分配内存的释放，最好使用 C++ 提供的手工分配系统。对于特定的情况，如自由内存出乎意料地持续降低时，最好直接调用 `collect()`。

2.5.7 重载赋值运算符

`GCPtr` 重载了两个 `operator=()`：一个是为了将新的地址赋给 `GCPtr`，另一个是为了将一个 `GCPtr` 指针赋给另一个 `GCPtr`。下面给出了这两个版本：

```
// Overload assignment of pointer to GCPtr.
template <class T, int size>
T * GCPtr<T, size>::operator=(T *t) {
    list<GCInfo<T> >::iterator p;

    // First, decrement the reference count
    // for the memory currently being pointed to.
    p = findPtrInfo(addr);
    p->refcount--;
    // Next, if the new address is already
    // existent in the system, increment its
    // count. Otherwise, create a new entry
    // for gclist.
    p = findPtrInfo(t);
    if(p != gclist.end())
        p->refcount++;
    else {
        // Create and store this entry.
        GCInfo<T> gcObj(t, size);
        gclist.push_front(gcObj);
    }

    addr = t; // store the address.
```

```

    return t;
}

// Overload assignment of GCPtr to GCPtr.
template <class T, int size>
GCPtr<T, size> & GCPtr<T, size>::operator=(GCPtr &rv) {
    list<GCInfo<T> >::iterator p;

    // First, decrement the reference count
    // for the memory currently being pointed to.
    p = findPtrInfo(addr);
    p->refcount--;

    // Next, increment the reference count of
    // the new address.
    p = findPtrInfo(rv.addr);
    p->refcount++; // increment ref count

    addr = rv.addr; // store the address.

    return rv;
}

```

第一个重载的 `operator=()` 处理了 `GCPtr` 指针在左、地址在右的赋值情况。例如，它可以处理如下的情况：

```

GCPtr<int> p;
// ...
p = new int(18);

```

在此，`new` 返回的地址赋给了 `p`。当这个操作发生时，就会调用 `operator=(T*t)`，并且将新地址传递给 `t`。首先，找到 `gclist` 中关于当前指向的内存的条目，并且其引用计数减 1。随后，搜索 `gclist` 查找新地址。如果能够找到的话，它的引用计数就会增 1。否则，就会为这个新地址建立新 `GCInfo` 对象，并且将这个对象加入到 `gclist`。最后，在调用对象的 `addr` 中存储这个新地址，并且返回这个地址。

赋值运算符的第二个重载，`operator=(GCPtr&rv)`，处理下面的赋值类型：

```

GCPtr<int> p;
GCPtr<int> q;
// ...
p = new int(88);
q = p;

```

在此，`p` 和 `q` 都是 `GCPtr` 指针，`p` 的值赋给了 `q`。这个版本与另一个版本的赋值运算符的运行方式很类似。首先，在 `gclist` 中找到了当前指向的内存的条目，其引用计数减 1。随后，搜索 `gclist` 来查找新地址，这个地址包含在 `rv.addr` 中，并且它的引用计数增 1。然后调用对象的 `addr` 字段被设置为包含在 `rv.addr` 中的地址。最后，返回右边的对象。从而允许进行连续的赋值，例如：

```

p = q = w = z;

```

关于赋值运算符的工作方式，还有很重要的一点需要说明。在本章的前面部分已经提到过，在技术上，可以在内存的引用计数减为 0 的时候，立即回收这块内存，但是这样做对每个指针操作都加入了额外的开销。这就是在赋值运算符的重载中，左操作数先前指向的内存的引用计数只是简单地减小，而没有执行另外动作的原因。因此，避免了与实际的内存释放以及对 `gclist` 的维护相关的开销。这些动作被推迟到垃圾回收器运行的时刻。这种方法使得使用了 `GCPtr` 的代码运行效率较高。同时还使得垃圾回收在尽量不影响系统性能的情况下进行。

2.5.8 GCPtr 的复制构造函数

由于必须通过跟踪每一个指针来分配内存，因此不能使用系统默认的复制构造函数。取而代之的是 `GCPtr` 必须定义它自己的复制构造函数，如下所示：

```
// Copy constructor.
GCPtr(const GCPtr &ob) {
    list<GCInfo<T> >::iterator p;

    p = findPtrInfo(ob.addr);
    p->refcount++; // increment ref count

    addr = ob.addr;
    arraySize = ob.arraySize;
    if(arraySize > 0) isArray = true;
    else isArray = false;
#ifdef DISPLAY
    cout << "Constructing copy.";
    if(isArray)
        cout << " Size is " << arraySize << endl;
    else
        cout << endl;
#endif
}
```

当需要对象的副本时，如对象作为参数传递给函数时、对象作为函数的返回值时、或者使用一个对象来初始化另一个对象时，就会调用类的复制构造函数。`GCPtr` 的复制构造函数复制保存在原始对象中的信息。并且它还增加了原始对象所指向的内存的引用计数。当这个副本超出作用域时，引用计数会减少。

实际上，通常并不需要由复制构造函数所执行的额外的工作，因为重载的赋值运算符在大多数情况下会恰当地维护垃圾回收链表。然而，在少数情况下还是需要复制构造函数，如在函数中分配内存并且返回指向这块内存的 `GCPtr` 的时候。

2.5.9 指针运算符和转换函数

因为 `GCPtr` 是一个指针类型，所以它必须重载指针运算符 `*` 和 `->`，还有索引运算符 `[]`。这些工作由这里显示的函数完成。尽管正是这些被重载的运算符的功能使得建立新的指针类型成为可能，但在此它们却非常简单：

```
// Return a reference to the object pointed
// to by this GCPtr.
```

```

T &operator*() {
    return *addr;
}

// Return the address being pointed to.
T *operator->() { return addr; }

// Return a reference to the object at the
// index specified by i.
T &operator[](int i) {
    return addr[i];
}

```

`operator*()`函数返回一个对象的引用,由调用 `GCPtr` 的 `addr` 字段指向这个对象。`operator->()`返回了包含在 `addr` 中的地址, `operator[]`返回了对索引指定的元素的引用。`operator[]`只能在指向已分配数组的 `GCPtr` 上使用。

如前所述,在此不支持指针运算。例如,没有为 `GCPtr` 重载 “++” 或者 “--” 运算符。这样做的原因是垃圾回收机制假定 `GCPtr` 指向已分配内存的开始。如果 `GCPtr` 可以增加,那么当这个指针被垃圾回收的时候, `delete` 使用的地址就会变得无效。

如果需要执行涉及到指针运算的操作,将有两个选择。首先,如果 `GCPtr` 指向一个已分配的数组,那么可以建立 `Iter` 来遍历数组。稍后将讲述这种方法。其次,可以使用 `GCPtr` 定义的 `T*`转换函数将 `GCPtr` 转换成一个普通的指针。这个函数如下所示:

```

// Conversion function to T *.
operator T*() { return addr; }

```

这个函数返回一个普通指针,这个指针指向存储在 `addr` 中的地址。这个函数可以如下使用:

```

GCPtr<double> gcp = new double(99.2);
double *p;

p = gcp; // now, p points to same memory as gcp
p++; // because p is a normal pointer, it can be incremented

```

在前面的示例中,由于 `p` 是一个普通指针,从而可以使用对其他指针使用的任何方法来操作它。当然,这个操作是否产生了有意义的结果,取决于您的应用程序。

对 `T*`的转换最主要的优点是,当与现有的、需要这种指针的代码一起工作时,可以使用 `GCPtr` 来取代普通的 C++ 指针。例如,考虑下面的代码:

```

GCPtr<char> str = new char[80];
strcpy(str, "this is a test");
cout << str << endl;

```

在此, `str` 是一个指向 `char` 的 `GCPtr` 指针,在 `strcpy()`的调用中用到了它。由于 `strcpy()`需要 `char*`类型的参数,因此自动调用 `GCPtr` 中对 `T*`的转换,因为在此情况下, `T` 是 `char`。在 `cout` 语句中使用 `str` 的时候,也会自动调用相同的转换。因此,转换函数使得 `GCPtr` 与现有的 C++ 函数和类无缝地整合在一起。

记住,这种转换返回的 `T*`指针既不参与也不影响垃圾回收。因此,即使常规的 C++ 指针

仍然指向它，已分配的内存也可以被释放。因此，应该明智且尽量少使用对 T^* 的转换。

2.5.10 begin()和 end()函数

下面所示的 `begin()` 和 `end()` 函数与 STL 中对应的函数很相似：

```
// Return an Iter to the start of the allocated memory.
Iter<T> begin() {
    int size;

    if(isArray) size = arraySize;
    else size = 1;

    return Iter<T>(addr, addr, addr + size);
}

// Return an Iter to one past the end of an allocated array.
Iter<T> end() {
    int size;

    if(isArray) size = arraySize;
    else size = 1;

    return Iter<T>(addr + size, addr, addr + size);
}
```

`begin()` 函数返回了一个 `Iter`，这个 `Iter` 指向 `addr` 指向的已分配数组的起始位置。`end()` 函数返回一个 `Iter`，这个 `Iter` 指向这个数组结尾的后一个位置。尽管在此并没有阻止指向单个对象的 `GCPtr` 调用这两个函数，但它们的目的是为了支持对已分配数组的操作。(为单个对象获取一个 `Iter` 并没有坏处，只是没有意义)。

2.5.11 shutdown()函数

如果一个程序建立了 `GCPtr` 的循环引用，那么当这个程序结束时，仍然存在需要被释放的动态分配的对象。这很重要，因为这些对象可能有需要调用的析构函数。`shutdown()` 函数用来处理这种情况。当建立第一个 `GCPtr` 时，这个函数就由 `atexit()` 注册，如前所述。这意味着当程序结束时，会调用这个函数。

`shutdown()` 函数如下所示：

```
// Clear gclist when program exits.
template <class T, int size>
void GCPtr<T, size>::shutdown() {

    if(gclistSize() == 0) return; // list is empty

    list<GCInfo<T> >::iterator p;

    for(p = gclist.begin(); p != gclist.end(); p++) {
        // Set all reference counts to zero
        p->refcount = 0;
    }
}
```

```

}

#ifdef DISPLAY
    cout << "Before collecting for shutdown() for "
          << typeid(T).name() << "\n";
#endif

collect();

#ifdef DISPLAY
    cout << "After collecting for shutdown() for "
          << typeid(T).name() << "\n";
#endif
}

```

首先，如果这个链表是空的(通常是这样的)，shutdown()只是简单地返回。否则，将仍然存在于 gclist 中的条目的引用计数设置为 0，然后调用 collect()。collect()释放任何引用计数为 0 的对象。因此，将引用计数设置为 0 可以确保释放所有对象。

2.5.12 两个实用函数

最后，GCPtr 定义了两个实用函数。第一个是 gclistSize()函数，这个函数返回当前 gclist 中拥有的条目数量。第二个是 showlist()函数，这个函数显示 gclist 的内容。这两个函数对于实现垃圾回收指针类型都不是必须的，但是，如果您想要观察垃圾回收器的运行情况，它们就有用了。

2.6 GCInfo

在 gclist 中的垃圾回收列表包含了 GCInfo 类型的对象。GCInfo 类如下所示：

```

// This class defines an element that is stored
// in the garbage collection information list.
//
template <class T> class GCInfo {
public:
    unsigned refcount; // current reference count

    T *memPtr; // pointer to allocated memory

    /* isArray is true if memPtr points
       to an allocated array. It is false
       otherwise. */
    bool isArray; // true if pointing to array

    /* If memPtr is pointing to an allocated
       array, then arraySize contains its size */
    unsigned arraySize; // size of array

    // Here, mPtr points to the allocated memory.

```

```

// If this is an array, then size specifies
// the size of the array.
GCInfo(T *mPtr, unsigned size=0) {
    refcount = 1;
    memPtr = mPtr;
    if(size != 0)
        isArray = true;
    else
        isArray = false;

    arraySize = size;
}
};

```

如前所述，每一个 GCInfo 对象都在 memPtr 中存储了一个指向已分配内存的指针，并在 refcount 中存储了与这块内存相关的引用计数。如果 memPtr 指向的内存包含了数组，那么当 GCInfo 对象建立时，必须指定这个数组的长度。在此情况下，isArray 被设置为 true，数组的长度将存储在 arraySize 中。

GCInfo 对象存储在一个 STL list 中。为了能够搜索这个链表，必须定义 operator==()，如下所示：

```

// Overloading operator== allows GCInfos to be compared.
// This is needed by the STL list class.
template <class T> bool operator==(const GCInfo<T> &ob1,
                                   const GCInfo<T> &ob2) {
    return (ob1.memPtr == ob2.memPtr);
}

```

只有在两个对象的 memPtr 字段相同时，它们才会相等。为了使得 GCInfo 存储在 STL 列表中，可能还需要重载其他运算符，这取决于您所使用的编译器。

2.7 Iter

Iter 类实现了一个类似于迭代器的对象，可以使用它遍历已分配数组的元素。在技术上 Iter 并非必须存在，因为 GCPtr 可以转换为一个基本类型的普通指针，但是 Iter 具有两个优点。首先，它可以使用遍历 STL 容器内容的方式来遍历已分配的数组。因此，使用 Iter 的语法是为人熟知的。其次，Iter 不允许越界访问。因此，相对于普通的指针，使用 Iter 比较安全。然而需要知道，Iter 没有参与垃圾回收。因此，如果将 Iter 作为基础的 GCPtr 超出了作用域，那么 GCPtr 所指向的内存就会被释放，无论 Iter 是否还需要它。

Iter 是一个模板类，其定义如下：

```
template <class T> class Iter {
```

Iter 指向的数据类型通过 T 传递。

Iter 定义了如下的实例变量：

```
T *ptr;    // current pointer value
```

```

T *end;    // points to element one past end
T *begin;  // points to start of allocated array
unsigned length; // length of sequence

```

Iter 当前所指的地址保存在 ptr 中。数组的起始地址保存在 begin 中，数组结尾的后一个元素的地址保存在 end 中。动态数组的长度保存在 length 中。

Iter 定义了这里所显示的两个构造函数。第一个是默认的构造函数。第二个建立了一个 Iter，并给出了 ptr 的初始值，另外还有指向数组起始位置和结束位置的指针。

```

Iter() {
    ptr = end = begin = NULL;
    length = 0;
}

Iter(T *p, T *first, T *last) {
    ptr = p;
    end = last;
    begin = first;
    length = last - first;
}

```

为了被本章给出的垃圾回收器代码使用，ptr 的初始值总是等于 begin。然而，可以随意地建立 Iter，在其中 ptr 的初始值是不同的值。

为了使得 Iter 像普通的指针那样，它重载了 * 和 -> 指针运算符，以及数组索引运算符 []，如下所示：

```

// Return value pointed to by ptr.
// Do not allow out-of-bounds access.
T &operator*() {
    if( (ptr >= end) || (ptr < begin) )
        throw OutOfRangeExc();
    return *ptr;
}

// Return address contained in ptr.
// Do not allow out-of-bounds access.
T *operator->() {
    if( (ptr >= end) || (ptr < begin) )
        throw OutOfRangeExc();
    return ptr;
}

// Return a reference to the object at the
// specified index. Do not allow out-of-bounds
// access.
T &operator[](int i) {
    if( (i < 0) || (i >= (end-begin)) )
        throw OutOfRangeExc();
    return ptr[i];
}

```

*运算符返回动态数组中当前指向元素的引用。->运算符返回当前指向元素的地址。[] 返回

指定索引处的元素的引用。注意这些操作都不允许越界访问。如果尝试这样做，就会抛出 `OutOfRangeExc` 异常。

`Iter` 定义了不同的指针算术运算符，如 `++`、`--` 等，用来递增或者递减 `Iter`。使用这些运算符可以遍历一个动态数组。考虑到速度的因素，这些算术运算符本身没有执行范围检查。然而，任何访问边界外元素的尝试都会导致异常，从而阻止了越界错误。`Iter` 定义了关系运算符。指针算术运算和关系运算的函数都相当直接并且易于理解。

`Iter` 还定义了一个名为 `size()` 的实用函数，它返回 `Iter` 所指数组的长度。

如前所述，在 `GCPtr` 中，对于每个 `GCPtr` 的实例，`Iter<T>` 都被类型定义为 `GCIterator`，从而简化了迭代器的声明。这意味着您可以使用类型名 `GCIterator` 来为任何 `GCPtr` 获取 `Iter`。

2.8 如何使用 GCPtr

`GCPtr` 的使用相当容易。首先，包含文件 `gc.h`。然后，声明一个 `GCPtr` 对象，并且指定它所指向的数据的类型。例如，为了声明一个名为 `p` 的、指向 `int` 的 `GCPtr` 对象，可以使用下面的声明：

```
GCPtr<int> p; // p can point to int objects
```

随后，使用 `new` 动态分配内存，并将 `new` 返回的指针赋给 `p`，如下所示：

```
p = new int; // assign p the address of an int
```

可以使用这样的赋值操作来给已分配内存赋一个值：

```
*p = 88; // give that int a value
```

当然，可以将前面的 3 个语句合并在一起：

```
GCPtr<int> p = new int(88); // declare and initialize
```

可以获取 `p` 所指位置的内存的值，如下所示：

```
int k = *p;
```

正如前面示例所显示的那样，通常使用 `GCPtr` 的方式类似于使用 C++ 的普通指针的方式。惟一区别就是当您不再需要此指针时，不需要删除它。当为这个指针分配的内存不再需要的时候，会自动释放它。

下面是组合了前面所示片断的完整程序：

```
#include <iostream>
#include <new>
#include "gc.h"

using namespace std;

int main() {
    GCPtr<int> p;

    try {
```

```

    p = new int;
} catch (bad_alloc exc) {
    cout << "Allocation failure!\n";
    return 1;
}

*p = 88;

cout << "Value at p is: " << *p << endl;

int k = *p;

cout << "k is " << k << endl;

return 0;
}

```

当开启显示选项时，这个程序的输出如下所示。(可以在 `gc.h` 中定义 `DISPLAY` 来观察垃圾回收器的运行):

```

Constructing GCPtr.
Value at p is: 88
k is 88
GCPtr going out of scope.
Before garbage collection for gclist<int, 0>:
memPtr      refcount      value
{002F12C0}    0            88
{00000000}    0            ---

Deleting: 88
After garbage collection for gclist<int, 0>:
memPtr      refcount      value
-- Empty --

```

当程序结束时，`p` 超出作用域。这将调用它的析构函数，从而使得 `p` 所指内存的引用计数递减。由于 `p` 是这块内存惟一的指针，因此这个操作使得引用计数变为 0。随后，`p` 的析构函数调用 `collect()`，`collect` 扫描 `gclist`，查找是否有引用计数为 0 的条目。由于先前与 `p` 相关条目的引用计数为 0，因此它所指的内存就会被释放。

2.8.1 处理分配异常

正如前面程序所显示的那样，由于垃圾回收器没有改变通过 `new` 分配内存的方式，因此可以用通常的方法捕获 `bad_alloc` 异常来处理分配失败。(当 `new` 失败时，就会抛出 `bad_alloc` 类型的异常)。当然，前面的程序不会用尽内存，从而实际上不需要 `try/catch` 块，但是实际的程序可能会耗尽堆。因此，您应该检测这种异常。

通常，在使用垃圾回收时，对 `bad_alloc` 异常最佳的响应方式是调用 `collect()` 来回收任何不再使用的内存，然后重新尝试分配内存。在本章稍后显示的加载-测试程序中采用了这种技术。也可以在您的程序中使用这种技术。

2.8.2 一个更有趣的示例

在此有一个更有趣的示例，显示了在程序结束之前 GCPtr 超出作用域的效果：

```
// Show a GCPtr going out of scope prior to the end
// of the program.
#include <iostream>
#include <new>
#include "gc.h"

using namespace std;

int main() {
    GCPtr<int> p;
    GCPtr<int> q;

    try {
        p = new int(10);
        q = new int(11);

        cout << "Value at p is: " << *p << endl;
        cout << "Value at q is: " << *q << endl;

        cout << "Before entering block.\n";

        // Now, create a local object.
        { // start a block
            GCPtr<int> r = new int(12);
            cout << "Value at r is: " << *r << endl;
        } // end the block, causing r to go out of scope

        cout << "After exiting block.\n";

    } catch(bad_alloc exc) {
        cout << "Allocation failure!\n";
        return 1;
    }

    cout << "Done\n";

    return 0;
}
```

当开启显示选项时，整个程序的输出如下：

```
Constructing GCPtr.
Constructing GCPtr.
Value at p is: 10
Value at q is: 11
Before entering block.
Constructing GCPtr.
Value at r is: 12
```

GCPtr going out of scope.

Before garbage collection for gclist<int, 0>:

memPtr	refcount	value
[002F31D8]	0	12
[002F12F0]	1	11
[002F12C0]	1	10
[00000000]	0	---

Deleting: 12

After garbage collection for gclist<int, 0>:

memPtr	refcount	value
[002F12F0]	1	11
[002F12C0]	1	10

After exiting block.

Done

GCPtr going out of scope.

Before garbage collection for gclist<int, 0>:

memPtr	refcount	value
[002F12F0]	0	11
[002F12C0]	1	10

Deleting: 11

After garbage collection for gclist<int, 0>:

memPtr	refcount	value
[002F12C0]	1	10

GCPtr going out of scope.

Before garbage collection for gclist<int, 0>:

memPtr	refcount	value
[002F12C0]	0	10

Deleting: 10

After garbage collection for gclist<int, 0>:

memPtr	refcount	value
-- Empty --		

下面仔细地分析这个程序及其输出。首先，注意在 main() 的开始创建了 p 和 q，但是直到输入 r 的代码块才创建 r。在 C++ 中，直到进入局部变量所在的代码块时，才会创建局部变量。当创建 r 时，它所指的内存被赋予了初始值 12。然后显示这个值，代码块结束。这使得 r 超出了作用域，意味着调用了它的析构函数。从而 gclist 中 r 的引用计数减小到 0。然后调用 collect() 来回收垃圾。

由于开启了显示选项，当 collect() 开始时，就会显示 gclist 的内容。注意它有 4 个条目。第一个条目先前与 r 链接。注意其引用计数为 0，说明由 memPtr 字段指的内存不再被任何程序元素使用。后面的两个条目仍然是活动的，并且与 p 和 q 链接。由于仍然使用它们，因此在此时不会释放它们所指的内存。最后一个条目代表了一个空指针，指向 p 和 q 创建时最初指向的条目。由于不再使用它，因此，collect() 将会把它从链表中移除。(当然，当移除 null 指针时，不会释放内存)。

由于没有其他的 GCPtr 指向与 r 相同的内存，这块内存就可以释放了，Deleting: 12 行证

明了这一点。当完成这些动作的时候，这个代码块之后的程序继续执行。最后，`p` 和 `q` 在程序结束时超出了作用域，释放它们所指的内存。在此情况下，首先调用 `q` 的析构函数，意味着首先将它回收。最后销毁 `p`，`gclist` 为空。

2.8.3 对象的分配和丢弃

当某块内存的引用计数变为 0 的时候(意味着没有 `GCPtr` 指向它)，这块内存就要被回收，理解这一点很重要。并不需要原先指向这个对象的 `GCPtr` 一定要超出作用域。因此，可以使用一个 `GCPtr` 对象指向任意数量的已分配的对象，只需要将这个 `GCPtr` 赋予一个新的值。被丢弃的内存最终会被回收。例如：

```
// Allocate and discard objects.
#include <iostream>
#include <new>
#include "gc.h"

using namespace std;

int main() {
    try {
        // Allocate and discard objects.
        GCPtr<int> p = new int(1);
        p = new int(2);
        p = new int(3);
        p = new int(4);

        // Manually collect unused objects for
        // demonstration purposes.
        GCPtr<int>::collect();

        cout << "*p: " << *p << endl;
    } catch(bad_alloc exc) {
        cout << "Allocation failure!\n";
        return 1;
    }

    return 0;
}
```

当开启显示选项时，程序的输出如下：

```
Constructing GCPtr.
Before garbage collection for gclist<int, 0>:
memPtr      refcount  value
[002F1310]    1         4
[002F1300]    0         3
[002F12D0]    0         2
[002F12A0]    0         1

Deleting; 3
```

```

Deleting: 2
Deleting: 1
After garbage collection for gclist<int, 0>:
memPtr      refcount      value
[002F1310]      1          4

*p: 4
GCPtr going out of scope.
Before garbage collection for gclist<int, 0>:
memPtr      refcount      value
[002F1310]      0          4

Deleting: 4
After garbage collection for gclist<int, 0>:
memPtr      refcount      value
-- Empty --

```

在这个程序中，`p` 是指向 `int` 的 `GCPtr`，被赋予了 4 个指向独立动态内存块的指针，每一块内存都使用不同的值进行了初始化。然后调用了 `collect()`，强制进行垃圾回收。注意 `gclist` 的内容：3 个条目被标记为不活跃，只有指向最后分配的内存的条目仍然在使用。随后，删除不再使用的条目。最终，程序结束，`p` 超出了作用域，最后的条目被删除。

注意，`p` 指向的前三块内存的引用计数为 0。这是由重载的赋值运算符的运行方式决定的。当给 `GCPtr` 赋予一个新地址时，它的原始值的引用计数减小。因此，每次给 `p` 赋予新整型数的地址时，旧地址的引用计数就会减小。

另外，由于 `p` 在声明的时候就被初始化，因此没有产生 `null` 指针条目，也没有将其放入到 `gclist` 中。记住，只有在声明 `GCPtr` 而不使用初始值时，才会建立 `null` 指针条目。

2.8.4 分配数组

如果使用 `new` 分配数组，那么必须在声明 `GCPtr` 所指的数组时指定它的大小，以通知 `GCPtr` 这一事实。例如，下面是分配 5 个 `double` 数组的方法：

```
GCPtr<double, 5> pda = new double[5];
```

有两个原因使得必须指定这个大小。首先，这样做会通知 `GCPtr` 构造函数，这个对象将指向一个已分配的数组，从而使得 `isArray` 字段为 `true`。当 `isArray` 为 `true` 的时候，`collect()` 函数使用 `delete[]` 释放内存，从而释放了动态分配的数组，而不是使用 `delete` 仅释放一个对象。因此，在示例中，当 `pda` 超出作用域时，使用了 `delete[]`，`pda` 的 5 个元素全部被释放。当分配类对象的数组时，确保释放正确数量的对象尤其重要。只有使用 `delete[]`，才能够确保调用每个对象的析构函数。

必须指定数组大小的第二个原因是为了在使用 `Iter` 遍历已分配数组时，阻止对数组元素的越界访问。当需要 `Iter` 时，数组的大小(存储在 `arraySize` 中)由 `GCPtr` 传递给 `Iter` 的构造函数。

要知道，在此并没有强制要求只能通过指向数组时指定的 `GCPtr` 来操作已分配的数组。您自己应该承担这个责任。

在您想要分配数组时，有两种方法可以访问它的元素。首先，可以给指向它的 `GCPtr` 做索引。其次，可以使用迭代器。以下介绍这两种方法。

1. 使用数组索引

下面程序创建的 `GCPtr` 指向具有 10 个元素的 `int` 数组。然后分配了这个数组，并将其初始化为值 0~9。最后通过给 `GCPtr` 做索引，显示这些值：

```
// Demonstrate indexing a GCPtr.
#include <iostream>
#include <new>
#include "gc.h"

using namespace std;

int main() {

    try {
        // Create a GCPtr to an allocated array of 10 ints.
        GCPtr<int, 10> ap = new int[10];

        // Give the array some values using array indexing.
        for(int i=0; i < 10; i++)
            ap[i] = i;

        // Now, show the contents of the array.
        for(int i=0; i < 10; i++)
            cout << ap[i] << " ";

        cout << endl;

    } catch(bad_alloc exc) {
        cout << "Allocation failure!\n";
        return 1;
    }

    return 0;
}
```

当关闭显示选项时，输出如下：

```
0 1 2 3 4 5 6 7 8 9
```

由于 `GCPtr` 模拟了普通的 C++ 指针，没有执行数组边界的检查，因此有可能会越界访问动态分配的数组。因此，在使用 `GCPtr` 访问数组时，要像使用普通的 C++ 指针访问数组那样小心。

2. 使用迭代器

尽管数组索引是遍历已分配的数组的方便方法，但它并不是您处理问题的惟一方法。对于许多应用程序，使用迭代器将是更好的选择，因为这样做可以防止越界的错误。对于 `GCPtr`，迭代器是 `Iter` 类型的对象。`Iter` 支持全部的指针运算，如++。它还允许迭代器像数组那样被索引。

在此使用迭代器修改前面的程序示例。获取 `GCPtr` 的迭代器的最方便方法是使用

GCIterator, 这是在 GCPtr 内的 typedef, GCPtr 自动绑定到通用类型 T:

```
// Demonstrate an iterator.
#include <iostream>
#include <new>
#include "gc.h"

using namespace std;

int main() {

try {
    // Create a GCPtr to an allocated array of 10 ints.
    GCPtr<int, 10> ap = new int[10];

    // Declare an int iterator.
    GCPtr<int>::GCIterator itr;

    // Assign itr a pointer to the start of the array.
    itr = ap.begin();

    // Give the array some values using array indexing.
    for(unsigned i=0; i < itr.size(); i++)
        itr[i] = i;

    // Now, cycle through array using the iterator.
    for(itr = ap.begin(); itr != ap.end(); itr++)
        cout << *itr << " ";

    cout << endl;

} catch(bad_alloc exc) {
    cout << "Allocation failure!\n";
    return 1;
} catch(OutOfRangeException exc) {
    cout << "Out of range access!\n";
    return 1;
}

return 0;
}
```

您可能会试图递增 itr, 以使得它所指的位置越过已分配数组的边界, 然后会试图在这个位置访问值。您将会看到, 这样做会抛出 OutOfRangeExc 异常。通常, 可以使用您喜欢的任意方式增加或者减小迭代器而不会导致异常。然而, 如果迭代器所指向的位置不在数组范围之内的话, 尝试在这个位置获取或者设置值都会导致越界错误。

2.8.5 使用具有类类型的 GCPtr

使用具有类类型的 GCPtr 与使用具有内建类型的 GCPtr 没有区别。例如, 在此由一个简短

的程序分配了 MyClass 的对象:

```
// Use GCPtr with a class type.
#include <iostream>
#include <new>
#include "gc.h"

using namespace std;

class MyClass {
    int a, b;
public:
    double val;

    MyClass() { a = b = 0; }

    MyClass(int x, int y) {
        a = x;
        b = y;
        val = 0.0;
    }

    ~MyClass() {
        cout << "Destructing MyClass(" <<
            a << ", " << b << ")\n";
    }

    int sum() {
        return a + b;
    }

    friend ostream &operator<<(ostream &strm, MyClass &obj);
};

// An overloaded inserter to display MyClass.
ostream &operator<<(ostream &strm, MyClass &obj) {
    strm << "(" << obj.a << " " << obj.b << ")";
    return strm;
}

int main() {
    try {
        GCPtr<MyClass> ob = new MyClass(10, 20);

        // Show value via overloaded inserter.
        cout << *ob << endl;

        // Change object pointed to by ob.
        ob = new MyClass(11, 21);
        cout << *ob << endl;

        // Call a member function through a GCPtr.
```

```

    cout << "Sum is : " << ob->sum() << endl;

    // Assign a value to a class member through a GCPtr.
    ob->val = 98.6;
    cout << "ob->val: " << ob->val << endl;

    cout << "ob is now " << *ob << endl;
} catch(bad_alloc exc) {
    cout << "Allocation error!\n";
    return 1;
}

return 0;
}

```

注意使用->运算符访问 MyClass 的成员的方式。记住，GCPtr 定义了一个指针类型，因此，对 GCPtr 执行的操作与其他任何类型指针的操作方式完全相似。

当关闭显示选项时，程序的输出如下：

```

(10 20)
(11 21)
Sum is : 32
ob->val: 98.6
ob is now (11 21)
Destructing MyClass(11, 21)
Destructing MyClass(10, 20)

```

特别注意最后两行，这是在垃圾回收时~MyClass()的输出。虽然只建立了一个 GCPtr 指针，然而分配了两个 MyClass 对象。垃圾回收链表中的条目表示了这两个对象。当销毁 ob 时，扫描 gclist 以找到引用计数为 0 的条目。在此情况下，会找到这样的两个条目，并删除它们所指的内存。

2.8.6 一个比较大的演示程序

下面的程序是一个比较大的示例，它体现了 GCPtr 的所有特性：

```

// Demonstrating GCPtr.
#include <iostream>
#include <new>
#include "gc.h"

using namespace std;

// A simple class for testing GCPtr with class types.
class MyClass {
    int a, b;
public:
    double val;

    MyClass() { a = b = 0; }
}

```

```

MyClass(int x, int y) {
    a = x;
    b = y;
    val = 0.0;
}

~MyClass() {
    cout << "Destructing MyClass(" <<
        a << ", " << b << ")\n";
}

int sum() {
    return a + b;
}

friend ostream &operator<<(ostream &strm, MyClass &obj);
};

// Create an inserter for MyClass.
ostream &operator<<(ostream &strm, MyClass &obj) {
    strm << "(" << obj.a << " " << obj.b << ")\n";
    return strm;
}

// Pass a normal pointer to a function.
void passPtr(int *p) {
    cout << "Inside passPtr(): "
        << *p << endl;
}

// Pass a GCPtr to a function.
void passGCPtr(GCPtr<int, 0> p) {
    cout << "Inside passGCPtr(): "
        << *p << endl;
}

int main() {

    try {
        // Declare an int GCPtr.
        GCPtr<int> ip;

        // Allocate an int and assign its address to ip.
        ip = new int(22);

        // Display its value.
        cout << "Value at *ip: " << *ip << "\n\n";

        // Pass ip to a function
        passGCPtr(ip);

        // ip2 is created and then goes out of scope
    }
}

```

```

{
    GCPtr<int> ip2 = ip;
}

int *p = ip; // convert to int * pointer'

passPtr(p); // pass int * to passPtr()

*ip = 100; // Assign new value to ip

// Now, use implicit conversion to int *
passPtr(ip);
cout << endl;

// Create a GCPtr to an array of ints
GCPtr<int, 5> iap = new int[5];

// Initialize dynamic array.
for(int i=0; i < 5; i++)
    iap[i] = i;

// Display contents of array.
cout << "Contents of iap via array indexing.\n";
for(int i=0; i < 5; i++)
    cout << iap[i] << " ";
cout << "\n\n";

// Create an int GCiterator.
GCPtr<int>::GCiterator itr;

// Now, use iterator to access dynamic array.
cout << "Contents of iap via iterator.\n";
for(itr = iap.begin(); itr != iap.end(); itr++)
    cout << *itr << " ";
cout << "\n\n";

// Generate and discard many objects
for(int i=0; i < 10; i++)
    ip = new int(i+10);

// Now, manually garbage collect GCPtr<int> list.
// Keep in mind that GCPtr<int, 5> pointers
// will not be collected by this call.
cout << "Requesting collection on GCPtr<int> list.\n";
GCPtr<int>::collect();

// Now, use GCPtr with class type.
GCPtr<MyClass> ob = new MyClass(10, 20);

// Show value via overloaded insertor.
cout << "ob points to " << *ob << endl;

```

```

// Change object pointed to by ob.
ob = new MyClass(11, 21);
cout << "ob now points to " << *ob << endl;

// Call a member function through a GCPtr.
cout << "Sum is : " << ob->sum() << endl;

// Assign a value to a class member through a GCPtr.
ob->val = 19.21;
cout << "ob->val: " << ob->val << "\n\n";

cout << "Now work with pointers to class objects.\n";

// Declare a GCPtr to a 5-element array
// of MyClass objects.
GCPtr<MyClass, 5> v;

// Allocate the array.
v = new MyClass[5];

// Get a MyClass GCiterator.
GCPtr<MyClass>::GCiterator mcItr;

// Initialize the MyClass array.
for(int i=0; i<5; i++) {
    v[i] = MyClass(i, 2*i);
}

// Display contents of MyClass array using indexing.
cout << "Cycle through array via array indexing.\n";
for(int i=0; i<5; i++) {
    cout << v[i] << " ";
}
cout << "\n\n";

// Display contents of MyClass array using iterator.
cout << "Cycle through array through an iterator.\n";
for(mcItr = v.begin(); mcItr != v.end(); mcItr++) {
    cout << *mcItr << " ";
}
cout << "\n\n";

// Here is another way to write the preceding loop.
cout << "Cycle through array using a while loop.\n";
mcItr = v.begin();
while(mcItr != v.end()) {
    cout << *mcItr << " ";
    mcItr++;
}
cout << "\n\n";

cout << "mcItr points to an array that is "

```

```

    << mcItr.size() << " objects long.\n";

// Find number of elements between two iterators.
GCPtr<MyClass>::GCIterator mcItr2 = v.end()-2;
mcItr = v.begin();
cout << "The difference between mcItr2 and mcItr is "
    << mcItr2 - mcItr;
cout << "\n\n";

// Can also cycle through loop like this.
cout << "Dynamically compute length of array.\n";
mcItr = v.begin();
mcItr2 = v.end();
for(int i=0; i < mcItr2 - mcItr; i++) {
    cout << v[i] << " ";
}
cout << "\n\n";

// Now, display the array backwards.
cout << "Cycle through array backwards.\n";
for(mcItr = v.end()-1; mcItr >= v.begin(); mcItr--)
    cout << *mcItr << " ";
cout << "\n\n";

// Of course, can use "normal" pointer to
// cycle through array.
cout << "Cycle through array using 'normal' pointer\n";
MyClass *ptr = v;
for(int i=0; i < 5; i++)
    cout << *ptr++ << " ";
cout << "\n\n";

// Can access members through a GCIterator.
cout << "Access class members through an iterator.\n";
for(mcItr = v.begin(); mcItr != v.end(); mcItr++) {
    cout << mcItr->sum() << " ";
}
cout << "\n\n";

// Can allocate and delete a pointer to a GCPtr
// normally, just like any other pointer.
cout << "Use a pointer to a GCPtr.\n";
GCPtr<int> *pp = new GCPtr<int>();
*pp = new int(100);
cout << "Value at **pp is: " << **pp;
cout << "\n\n";

// Because pp is not a garbage collected pointer,
// it must be deleted manually.
delete pp;
} catch(bad_alloc exc) {
    // A real application could attempt to free

```

```

        // memory by collect() when an allocation
        // error occurs.
        cout << "Allocation error.\n";
    }

    return 0;
}

```

当关闭显示选项时，程序输出如下：

```

Value at *ip: 22

Inside passGCPtr(): 22
Inside passPtr(): 22
Inside passPtr(): 100

Contents of iap via array indexing.
0 1 2 3 4

Contents of iap via iterator.
0 1 2 3 4

Requesting collection on GCPtr<int> list.
ob points to (10 20)
ob now points to (11 21)
Sum is : 32
ob->val: 19.21

Now work with pointers to class objects.
Destructing MyClass(0, 0)
Destructing MyClass(1, 2)
Destructing MyClass(2, 4)
Destructing MyClass(3, 6)
Destructing MyClass(4, 8)
Cycle through array via array indexing.
(0 0) (1 2) (2 4) (3 6) (4 8)

Cycle through array through an iterator.
(0 0) (1 2) (2 4) (3 6) (4 8)

Cycle through array using a while loop.
(0 0) (1 2) (2 4) (3 6) (4 8)

mcItr points to an array that is 5 objects long.
The difference between mcItr2 and mcItr is 3

Dynamically compute length of array.
(0 0) (1 2) (2 4) (3 6) (4 8)

Cycle through array backwards.
(4 8) (3 6) (2 4) (1 2) (0 0)

```

```
Cycle through array using 'normal' pointer
```

```
(0 0) (1 2) (2 4) (3 6) (4 8)
```

```
Access class members through an iterator.
```

```
0 3 6 9 12
```

```
Use a pointer to a GCPtr.
```

```
Value at **pp is: 100
```

```
Destructing MyClass(4, 8)
```

```
Destructing MyClass(3, 6)
```

```
Destructing MyClass(2, 4)
```

```
Destructing MyClass(1, 2)
```

```
Destructing MyClass(0, 0)
```

```
Destructing MyClass(11, 21)
```

```
Destructing MyClass(10, 20)
```

开启显示选项(也就是在 `gc.h` 中定义 `DISPLAY`), 试着编译并运行这个程序。然后浏览程序, 将输出与每条语句匹配。从而您会对垃圾回收器的运行方式有明确的认识。记住, 无论什么时候 `GCPtr` 超出作用域, 都会发生垃圾回收。这在程序的不同时刻发生, 例如, 当接收了 `GCPtr` 的副本的函数返回时, 在此情况下, 这个副本超出作用域, 从而发生垃圾回收。还要记住每一个类型的 `GCPtr` 都维护着自身的 `gclist`。因此, 从一个链表进行垃圾回收不会导致它从其他类型的链表回收。

2.8.7 加载测试

下面的程序通过重复地分配并丢弃对象直到自由内存耗尽来加载测试 `GCPtr`。当这些发生时, `new` 会抛出 `bad_alloc` 异常。在异常处理程序中, 显式地调用了 `collect()` 来回收不再使用的内存, 这个过程继续。可以在您自己的程序中使用相同的技术:

```
// Load test GCPtr by creating and discarding
// thousands of objects.
#include <iostream>
#include <new>
#include <limits>
#include "gc.h"
```

```
using namespace std;
```

```
// A simple class for load testing GCPtr.
```

```
class LoadTest {
```

```
    int a, b;
```

```
public:
```

```
    double n[100000]; // just to take up memory
```

```
    double val;
```

```
    LoadTest() { a = b = 0; }
```

```
    LoadTest(int x, int y) {
```

```
        a = x;
```

```
        b = y;
```



```

    val = 0.0;
}

friend ostream &operator<<(ostream &strm, LoadTest &obj);
};

// Create an inserter for LoadTest.
ostream &operator<<(ostream &strm, LoadTest &obj) {
    strm << "(" << obj.a << " " << obj.b << ")";
    return strm;
}

int main() {
    GCPtr<LoadTest> mp;
    int i;

    for(i = 1; i < 20000; i++) {
        try {
            mp = new LoadTest(i, i);
        } catch(bad_alloc xa) {
            // When an allocation error occurs, recycle
            // garbage by calling collect().
            cout << "Last object: " << *mp << endl;
            cout << "Length of gclist before calling collect(): "
                << mp.gclistSize() << endl;
            GCPtr<LoadTest>::collect();
            cout << "Length after calling collect(): "
                << mp.gclistSize() << endl;
        }
    }

    return 0;
}

```

这个程序的部分输出(关闭显示选项)如下。当然,您看到的确切输出可能会有所不同,因为您系统可用内存的数量与您使用的编译器可能不同。

```

Last object: (518 518)
Length of gclist before calling collect(): 518
Length after calling collect(): 1
Last object: (1036 1036)
Length of gclist before calling collect(): 518
Length after calling collect(): 1
Last object: (1554 1554)
Length of gclist before calling collect(): 518
Length after calling collect(): 1
Last object: (2072 2072)
Length of gclist before calling collect(): 518
Length after calling collect(): 1
Last object: (2590 2590)
Length of gclist before calling collect(): 518
Length after calling collect(): 1

```

```
Last object: (3108 3108)
Length of gclist before calling collect(): 518
Length after calling collect(): 1
Last object: (3626 3626)
Length of gclist before calling collect(): 518
Length after calling collect(): 1
```

2.8.8 一些限制

下面是使用 GCPtr 的一些限制:

(1) 不能创建全局的 GCPtr。程序的其余部分结束之后, 全局对象才会超出作用域。当全局的 GCPtr 超出作用域时, GCPtr 的析构函数调用 collect() 来试图释放不再使用的内存。问题是, 根据您的 C++ 编译器的实现方式, gclist 可能已经被销毁。在此情况下, 执行 collect() 会导致运行时错误。因此, GCPtr 只能在创建局部对象时使用。

(2) 当使用动态分配的数组时, 您在声明指向它的 GCPtr 时必须指定这个数组的大小。然而并没有机制强制这么做, 因此要小心。

(3) 不能通过显式地使用 delete 来释放 GCPtr 所指的内存。如果需要立即释放一个对象, 就可以调用 collect()。

(4) GCPtr 对象只能指向由 new 动态分配的内存。将任何其他类型的指针赋给 GCPtr 都会在 GCPtr 对象超出作用域时引发错误, 因为尝试释放从来没有分配过的内存。

(5) 最好避免循环指针引用, 原因在本章的前面部分已经描述过。尽管所有已分配的内存最终会被释放, 然而包含了循环引用的对象直到程序结束才会被释放, 而不是在其他程序元素不再使用它们的时候被释放。

2.9 试着完成下面的任务

可以容易地通过修改 GCPtr 来适应您的应用程序的需求。如前所述, 您想要尝试的改变之一是在达到某个标准时才试图回收垃圾, 如 gclist 达到了某个尺寸, 或者一定数量的 GCPtr 超出了作用域。

对 GCPtr 功能的一个有趣增强是重载 new, 从而可以在分配失败时自动回收垃圾。也可以在为 GCPtr 分配内存的时候不使用 new, 而使用 GCPtr 定义的工厂函数来代替。这样做可以让您更仔细地控制动态内存的分配, 但是使得分配过程从根本上不同于 C++ 内建的方法。

您可能会尝试使用其他方法来处理循环引用的问题。方法之一是实现弱引用的概念, 它不会阻止垃圾回收的发生。可以在需要循环引用的时候使用弱引用。

可能关于 GCPtr 最有趣的版本在第 3 章。在那里创建了一个多线程的版本, 当 CPU 空闲时, 垃圾回收就会自动进行。

第 3 章 C++ 中的多线程

在现代程序设计中，多线程变得越来越重要。原因之一是多线程可以使得程序更加充分地利用 CPU，从而使得编写的程序更加高效。另一个原因是对于处理事件驱动的代码，多线程是一种很自然的选择，在现在高度分布式的、网络化的、基于 GUI 的环境中，事件驱动代码非常普遍。当然，使用地最广泛的操作系统 Windows 支持多线程也是一个因素。无论因为什么原因，不断增加的多线程的使用改变了程序员对程序基本结构的认识。尽管 C++ 没有内建的对多线程程序的支持，但是它却非常适合这一领域。

由于多线程变得越来越重要，因此在本章将尝试使用 C++ 创建多线程程序。本章开发了两个多线程应用程序。第一个是线程控制面板，可以使用它来控制程序中线程的执行。这不仅是一个有趣的多线程示例，而且是一个在开发多线程应用程序时可以使用的实用工具。第二个程序创建了第 2 章垃圾回收器的修订版本，使得它在后台线程运行，从而说明了如何将多线程应用到实际的示例中。

本章还有另外一个目的：显示 C++ 是如何熟练地与操作系统直接对接。一些其他语言，如 Java，在您的程序和 OS 之间有一个处理层。对于某些类型的程序(如实时环境中使用的那些程序)而言，这个层会严重影响系统性能。与之形成鲜明对比的是，C++ 可以直接访问操作系统提供的底层功能。这也是 C++ 能够创造高性能代码的原因之一。

3.1 什么是多线程

在开始之前，有必要准确地定义术语多线程的含义。多线程是多任务的特殊形式。通常，有两种类型的多任务：基于进程和基于线程的多任务。进程本质上是正在执行的程序。因此，基于进程的多任务就是允许您的计算机同时运行两个或者更多程序的特性。例如，基于进程的多任务允许您在使用电子制表软件或者浏览 Internet 的同时运行文字处理程序。在基于进程的多任务中，程序是调度程序可以分派的最小代码单元。

线程是可执行代码的可分派单元。这个名称来源于“执行的线索”的概念。在基于线程的多任务的环境中，所有进程有至少一个线程，但是它们可以具有多个任务。这意味着单个程序可以并发执行两个或者多个任务。例如，文本编辑器可以在打印文本的同时格式化文本，只要这两个动作是被两个独立的线程执行。基于进程的多任务与基于线程的多任务之间的区别可以归纳如下：基于进程的多任务处理程序的并发执行，基于线程的多任务处理相同程序的不同片断的并发执行。

在前面的讨论中，需要明确：只有在多 CPU 的系统中，才可能有真正的并发执行，在那里每个进程或者线程可以不受限制地访问 CPU。对于单个 CPU 的系统(在当前使用的系统中，这是主流)，仅能够在表面上做到并发执行。在单个 CPU 的系统中，每个进程或者线程都接收一部分 CPU 时间，时间的数量由几个因素来确定，包括进程或线程的优先级。尽管大多数计

计算机并没有真正意义上的并发执行，但是在编写多线程应用程序的时候，您应该假定它确实有并发能力。这是因为您不能够知道单个线程执行的确切顺序，或者它们是否能够按照相同的顺序执行两次。因此，最好假定程序确实是在并发执行。

多线程对程序结构的改变

多线程改变了程序的基本结构。不同于按照严格的线性方式执行的单线程程序，多线程程序并发地执行它自身的各个部分。这样，所有的多线程程序都包含了相似的元素。因此，多线程程序的主要问题是管理线程之间的交互。

如前所述，所有的进程都至少包含一个执行线程，称之为主线程。主线程在程序开始时创建。在多线程程序中，主线程创建一个或者多个子线程。因此，每个多线程的进程都以一个执行线程开始，然后创建一个或者多个附加的线程。在设计合理的程序中，每个线程都代表一个逻辑上独立的活动单元。

多线程的主要优点是可以让您编写非常高效的程序，因为它使得您可以利用大多数程序都具有的空闲时间。大多数的 I/O 设备，无论是网络端口、磁盘驱动器还是键盘，速度都比 CPU 慢很多。通常，程序将主要的执行时间都花费在等待接收或者发送数据上。通过谨慎地使用多线程，您的程序可以在空闲的时候执行另一个任务。例如，当程序的一部分通过 Internet 发送文件时，另一个部分可以读取键盘的输入，还有一个部分可以将下一步要发送的数据块缓存。

3.2 为什么 C++ 没有内建支持多线程

C++ 没有包含任何对多线程应用程序的内建的支持。相反，它依赖于操作系统提供这个特性。考虑到 Java 和 C# 都提供了内建的多线程支持，您会很自然地问，为什么 C++ 没有提供，答案是因为效率、控制以及 C++ 适用的应用程序的范围。让我们逐一分析。

由于没有内建对多线程的支持，因此 C++ 没有尝试定义一种“万能的”解决方案。相反，C++ 允许您直接使用操作系统提供的多线程特性。这种方法意味着您的程序可以使用执行环境支持的、最高效的方法来实现多线程。由于许多的多任务环境提供了对多线程丰富的支持，因此能够访问这些支持对于创建高性能的多线程程序至关重要。

使用操作系统的函数来支持多线程使得可以全面地使用执行环境提供的控制。考虑 Windows 环境。它提供了多组线程相关函数来有条理地控制线程的创建和管理。例如，Windows 有多种方法来控制对共享资源的访问，共享资源包括信号、互斥体、事件对象、可等待定时器以及临界区。由于操作系统能力的不同，很难将这种灵活性设计到一门语言中。因此，对于多线程语言层次的支持通常意味着仅提供特性的“最小公倍数”。通过 C++，您可以访问操作系统提供的所有特性。当编写高性能的代码时，这是非常重要的优点。

C++ 是为所有类型的程序设计类型设计的，从嵌入式系统(在执行环境中没有操作系统)到高度分布的、基于 GUI 的终端用户应用程序以及介于二者之间的一切程序。因此，C++ 不能够对它的执行环境加入明显的限制。内建的对多线程的支持将会从根本上将 C++ 限制在那些支持多线程的环境中，从而阻止了在不使用线程的环境中开发软件时使用 C++。

在最后的分析中，没有内建的对多线程的支持是 C++ 的一个主要优点，因为这样可以使用对目标执行环境最高效的方式编写程序。记住，C++ 的功能无处不在。多线程的情况很明显是

一种“简单就好”的情况。

3.3 选用什么样的操作系统和编译器

由于 C++ 依赖于操作系统提供对多线程程序设计的支持, 因此在本章有必要选择一种操作系统作为多线程应用程序的平台。由于 Windows 是世界上使用最广泛的操作系统, 因此本章使用了这个操作系统。然而, 许多信息都可以适用于任何支持多线程的 OS。

由于在设计 Windows 程序时, Visual C++ 是应用最广泛的编译器, 它也就是本章示例使用的编译器。在下面的部分中这个重要性很明显。然而, 如果您使用了其他编译器, 就很容易修改这些代码来适应它。

提示:

本章的示例假定读者具有 Windows 程序设计的基本知识。

3.4 Windows 线程函数概述

Windows 提供了多组支持多线程的应用程序接口(API)函数。许多读者已经对 Windows 提供的多线程函数有一定程度的了解, 但是对于那些不熟悉这些的读者, 本章提供了这些函数的概述。记住, Windows 提供了许多其他的基于多线程的函数, 这些函数需要您自己去探索。

为了使用 Windows 的多线程函数, 必须在程序中包含 <Windows.h>。

3.4.1 线程的创建和终止

Windows API 提供了 CreateThread() 函数来创建一个线程。其原型如下所示:

```
HANDLE CreateThread(LPSECURITY_ATTRIBUTES secAttr,
                    SIZE_T stackSize,
                    LPTHREAD_START_ROUTINE threadFunc,
                    LPVOID param,
                    DWORD flags,
                    LPDWORD threadID);
```

在此, secAttr 是一个用来描述线程的安全属性的指针。如果 secAttr 是 NULL, 就会使用默认的安全描述符。

每个线程都具有自己的堆栈。可以使用 stackSize 参数来按字节指定新线程堆栈的大小。如果这个整数值为 0, 那么这个线程堆栈的大小与创建它的线程相同。如果需要的话, 这个堆栈可以扩展。(通常使用 0 来指定线程堆栈的大小)。

每个线程都在创建它的进程中通过调用线程函数来开始执行。线程的执行一直持续到线程函数返回。这个函数的地址(也就是线程的入口点)在 threadFunc 中指定。每个线程函数都必须具有这样的原型:

```
DWORD WINAPI threadfunc(LPVOID param);
```

需要传递给新线程的任何参数都在 CreateThread() 的 param 中指定。线程函数在它的参数中

接收这个 32 位的值。这个参数可以用作任何目的。函数返回它的退出状态。

参数 `flags` 确定了线程的执行状态。如果它是 0，线程会立即执行。如果是 `CREATE_SUSPEND`，线程则以挂起状态创建并等待执行。(可以通过调用 `ResumeThread()` 来开始执行，稍后讨论)。

与线程相关的标识符以 `threadID` 所指向的长整型返回。

如果成功，函数则向线程返回一个句柄。如果失败，则返回 `NULL`。可以通过调用 `CloseHandle()` 来显式销毁这个线程。否则，会在父进程结束时自动销毁它。

如前所述，当线程的入口函数返回时终止执行线程。进程也可以使用 `TerminateThread()` 或者 `ExitThread()` 来手动终止线程，这两个函数的原型如下：

```
BOOL TerminateThread(HANDLE thread, DWORD status);  
VOID ExitThread(DWORD status);
```

对于 `TerminateThread()`，`thread` 是即将终止的线程的句柄。`ExitThread()` 只能用来终止调用了 `ExitThread()` 的线程。对于两个函数而言，`status` 是终止状态。`TerminateThread()` 如果成功，则会返回非 0 值，否则返回 0。

调用 `ExitThread()` 在功能上等价于允许线程函数正常返回。这意味着堆栈会正确地重新设置。当使用 `TerminateThread()` 结束线程时，线程会立刻终止，而不会执行任何特定的清理任务。另外，`TerminateThread()` 可能会停止正在执行重要操作的线程。为此，当入口函数返回时，通常最好(也是最容易的)让线程正常终止。

3.4.2 Visual C++ 对 `CreateThread()` 和 `ExitThread()` 的替换

尽管 `CreateThread()` 和 `ExitThread()` 是用来创建并终止线程的 Windows API 函数，我们在本章并不会使用它们。原因是在 Visual C++ 中(其他的 Windows 兼容的编译器也可能有这个问题)使用这两个函数时，会导致内存泄漏，丢失少量的内存。对于 Visual C++，如果多线程程序利用了 C/C++ 标准库函数并使用了 `CreateThread()` 和 `ExitThread()`，就会丢失少量的内存。(如果您的程序没有使用 C/C++ 的标准库，就不会发生这样的内存丢失)。为了避免这种情况，必须使用 Visual C++ 运行库中定义的函数来开始和终止线程，而不是使用由 Win32 API 指定的函数。这些函数类似于 `CreateThread()` 和 `ExitThread()`，但是不会产生内存泄漏。

提示：

如果使用非 Visual C++ 的编译器，如果需要的话，检查它的文档来确定是否需要忽略 `CreateThread()` 和 `ExitThread()`，以及如何做到这一点。

Visual C++ 用 `_beginthreadex()` 和 `_endthreadex()` 来取代 `CreateThread()` 和 `ExitThread()`。这两个函数都需要头文件 `<process.h>`。下面是 `_beginthreadex()` 函数的原型：

```
uintptr_t _beginthreadex(void *secAttr, unsigned stackSize,  
                        unsigned (__stdcall *threadFunc)(void *),  
                        void *param, unsigned flags,  
                        unsigned *threadID);
```

正如您看到的那样，`_beginthreadex()` 的参数类似于 `CreateThread()` 的参数。另外，这些参数与 `CreateThread()` 指定的参数有相同的含义。`secAttr` 是一个用来描述线程安全性属性的指针。

然而，如果 `secAttr` 为 `NULL`，则会使用默认的安全描述符。新线程堆栈的大小由 `stackSize` 参数按字节数传递。如果这个值为 0，那么这个线程堆栈的大小与进程中创建它的主线程的大小相同。

线程函数的地址(也就是线程的入口点)在 `threadFunc` 中指定。对于 `_beginthreadex()`，线程函数的原型如下：

```
unsigned_stdcall threadfunc(void *param)
```

这个原型在功能上等价于 `CreateThread()` 的线程函数的原型，但是它使用了不同的类型名称。想要传递给新线程的任何参数都在 `param` 参数中指定。

`flags` 参数确定线程的执行状态。如果 `flags` 参数为 0，线程会立即开始执行。如果 `flags` 参数为 `CREATE_SUSPEND`，则以挂起状态创建线程。(可以调用 `ResumeThread()` 来开始它)。与线程相关的标识符以 `threadID` 指向的 `double word` 返回。

如果成功，则这个函数返回一个线程的句柄；如果失败，则返回 0。类型 `uintptr_t` 指定了可以拥有指针或者句柄的 Visual C++ 类型。

`_endthreadex()` 的原型如下：

```
void _endthreadex(unsigned status);
```

它的功能就像 `ExitThread()` 那样，停止线程并返回 `status` 中指定的退出代码(exit code)。

由于 Windows 下使用最广泛的编译器是 Visual C++，因此本章示例将使用 `_beginthreadex()` 和 `_endthreadex()` 而不是使用它们的等价的 API 函数。如果您使用了非 Visual C++ 的编译器，那么只需要用 `CreateThread()` 和 `EndThread()` 来替代这两个函数。

当使用 `_beginthreadex()` 和 `_endthreadex()` 时，必须记住链接多线程库。这随编译器的不同而不同。在此有一些示例。当使用 Visual C++ 的命令行编译器时，包括 `-MT` 选项。为了在 Visual C++ 6 IDE 中使用多线程库，首先要激活“Project | Settings”属性页。然后选择“C/C++”选项卡。接着在“Category”下拉列表框中选择“Code Generation”，然后在“Use Runtime Library”下拉列表框中选择“Multithreaded”。对于 Visual C++ 7 .NET IDE，选择“Project | Properties”。然后选择“C/C++”条目，并加亮显示“Code Generation”。最后，将“Multithreaded”选择为运行库。

3.4.3 线程的挂起和恢复

线程的执行可以通过调用 `SuspendThread()` 来挂起。可以通过调用 `ResumeThread()` 来恢复它。这两个函数的原型如下：

```
DWORD SuspendThread(HANDLE hThread);
```

```
DWORD ResumeThread(HANDLE hThread);
```

两个函数都通过 `hThread` 来传递线程的句柄。

每个执行的线程都有与其相关的挂起计数。如果这个计数为 0，那么不会挂起线程。如果为非 0 值，则线程就会处于挂起状态。每次调用 `SuspendThread()` 都会增加这个计数。每次调用 `ResumeThread()` 都会减小这个挂起计数。挂起的线程只有在它的挂起计数达到 0 时才会恢复。因此，为了恢复一个挂起的线程，对 `ResumeThread()` 的调用次数必须与对 `SuspendThread()` 的调

用次数相等。

这两个函数都返回线程先前的挂起计数，如果发生错误，返回值为 -1。

3.4.4 改变线程的优先级

在 Windows 中，每个线程都与一个优先级设置相关。线程的优先级决定了线程接收的 CPU 时间的多少。低优先级的线程接收比较少的时间，高优先级的线程接收比较多的时间。当然，线程接收的 CPU 时间的多少对于它的执行性能以及它与系统中当前执行的其他线程之间的交互有着深远的影响。

在 Windows 中，线程优先级的设置是两个值的组合：进程总体的优先级类别以及相对于这个优先级类别的各个线程的优先级设置。也就是说，线程实际的优先级由进程的优先级类别和各个线程的优先级的组合来确定。后面会逐一讲述。

1. 优先级类别

在默认情况下，进程具有普通的优先级类别，大多数程序在其执行的声明周期内保持这个普通的优先级类别。尽管在本章的示例中没有改变优先级类别，但是为了完整起见，在此给出了线程优先级类别的简单概况。

Windows 定义了 6 个优先级类别，相应的值以从高到低的顺序显示如下：

```
REALTIME_PRIORITY_CLASS
HIGH_PRIORITY_CLASS
ABOVE_NORMAL_PRIORITY_CLASS
NORMAL_PRIORITY_CLASS
BELOW_NORMAL_PRIORITY_CLASS
IDLE_PRIORITY_CLASS
```

在默认情况下，程序的优先级类别为 `NORMAL_PRIORITY_CLASS`。通常，您不需要改变程序的优先级类别。事实上，改变进程的优先级类别对于整个计算机系统的性能会有负面的影响。例如，如果您将一个程序的优先级类别增加到 `REALTIME_PRIORITY_CLASS`，它就会支配 CPU。对于某些特殊的应用程序，可能需要增加应用程序的优先级类别，但通常并不需要。如前所述，本章的应用程序没有改变优先级类别。

当确实需要改变程序的优先级类别时，可以调用 `SetPriorityClass()`。可以调用 `GetPriorityClass()` 来获取当前的优先级类别。这两个函数的原型如下：

```
DWORD GetPriorityClass(HANDLE hApp);

BOOL SetPriorityClass(HANDLE hApp, DWORD priority);
```

在此，`hApp` 是进程的句柄。`GetPriorityClass()` 返回应用程序的优先级类别，如果失败的话，返回 0。对于 `SetPriorityClass()`，`priority` 指定了进程的新优先级类别。

2. 线程优先级

对于给定的优先级类别，各个线程的优先级确定了它在进程内接收的 CPU 时间的多少。当线程第一次创建时，它具有普通的优先级，但是您可以改变线程的优先级——即使在它执行时。

可以通过调用 `GetThreadPriority()` 来获取线程的优先级设置。可以使用 `SetThreadPriority()` 来增加或者减小线程的优先级。这两个函数的原型如下：

```
BOOL SetThreadPriority(HANDLE hThread, int priority);

int GetThreadPriority(HANDLE hThread);
```

对于这两个函数而言，*hThread* 是线程的句柄。对于 `SetThreadPriority()`，*priority* 是新的优先级设置。如果发生错误，则返回值为 0；否则，返回非 0 值。`GetThreadPriority()` 会返回当前的优先级设置。优先级设置按照从高到低的顺序如表 3-1 所示。

表 3-1 优先级设置

线程优先级	值
THREAD_PRIORITY_TIME_CRITICAL	15
THREAD_PRIORITY_HIGHEST	2
THREAD_PRIORITY_ABOVE_NORMAL	1
THREAD_PRIORITY_NORMAL	0
THREAD_PRIORITY_BELOW_NORMAL	-1
THREAD_PRIORITY_LOWEST	-2
THREAD_PRIORITY_IDLE	-15

这些值相对于进程的优先级类别或增或减。通过组合进程的优先级类别和线程的优先级，Windows 向应用程序提供了 31 个不同的优先级设置的支持。

如果有错误发生，则 `GetThreadPriority()` 返回 `THREAD_PRIORITY_ERROR_RETURN`。

在大多数情况下，如果线程具有普通的优先级类别，那么可以随意地改变它的优先级设置，而不必担心会给整个系统的性能带来灾难性的影响。您将会看到，在下面部分开发的线程控制面板中，可以改变进程内线程的优先级设置(但是不能改变优先级类别)。

3.4.5 获取主线程的句柄

主线程的执行是可以控制的。为此，需要获取它的句柄。做到这一点最简单的方法是调用 `GetCurrentThread()`，其原型如下：

```
HANDLE GetCurrentThread(void);
```

这个函数返回当前线程的伪句柄(pseudohandle)。之所以称之为伪句柄，是因为它是一个预定义的值，总是引用当前的线程，而不是引用指定的调用线程。然而，它能够用在任何可以使用普通线程处理的地方。

3.4.6 同步

在使用多线程或多进程时，有时需要调整两个或者多个线程(或者进程)之间的活动。这个过程称为同步。当两个或者多个线程需要访问共享资源，而这个共享资源在同一时刻只能由一个线程使用时，就需要使用同步。例如，当一个线程在写文件时，在此时必须阻止另一个线程

也这么做。同步的另一个原因是有时线程需要等待由另一个线程引发的事件。在此情况下，必须采取某种措施将第一个线程保持挂起状态，直到这个事件发生。随后等待的线程必须恢复执行。

通常某个任务会处于两种状态。首先，它可能正在执行(或者在获得它的时间段时就开始执行)。另外，任务可能被阻塞，等待某个资源或者事件。在此情况下其执行被挂起，直到所需的资源可以使用或者所等待的事件发生。

如果您对于同步问题或者它的常用解决方案(信号量)不熟悉，下面的部分将对此进行讨论。

1. 理解同步问题

Windows 必须提供某种特殊的服务来允许对共享资源的访问同步，因为如果没有操作系统的协助，进程或者线程就没有办法得知它是否在单独访问某个资源。为了理解这个问题，假定您在为一个没有提供任何同步支持的多任务操作系统编写程序，并且假定您具有两个并发执行的线程 A 和 B，它们都不时地访问某个资源 R(如磁盘文件)，这个资源在某个时刻只能被一个线程访问。为了在一个线程使用这个资源时阻止另一个线程访问它，您尝试了下面的解决方案。首先，创建了一个初始化为 0 并且两个线程都可以访问的变量，名为 flag。然后，在使用访问 R 的每段代码之前，等待 flag 被清 0，然后设置 flag，访问 R，最后将 flag 清 0。也就是说，在每个线程访问 R 之前，执行如下的代码：

```
while(flag) ; // wait for flag to be cleared
flag = 1; // set flag

// ... access resource R ...

flag = 0; // clear the flag
```

这段代码隐含的概念是，如果设置了 flag，则两个线程都不能够访问 R。从概念上讲，这种方法是正确的解决方案。然而，实际上它远远没有达到要求，原因很简单：它并非总是有效！让我们看一下原因。

使用刚才给定的代码，有可能两个进程同时访问 R。while 循环在本质上执行重复的加载和比较 flag 上的指令。换句话说，它一直在测试 flag 的值。当 flag 被清 0 的时候，代码的下一行将设置 flag 的值。问题在于，这两个操作有可能在两个不同的时间段执行。在两个时间段之间，flag 的值有可能被另一个线程访问，从而 R 被两个线程同时访问。为了理解这一点，假定线程 A 进入 while 循环，发现 flag 为 0，这是访问 R 的绿灯。然而，在将 flag 设置为 1 之前，其时间段用尽，线程 B 恢复执行。如果 B 执行了它的 while，它也发现 flag 没有被设置，并且认为它可以安全地访问 R。然而，当 A 重新开始时，它也会访问 R。问题的关键在于对 flag 的测试和设置没有包含在一个连续的操作中，而是可以被分为两个部分，正如刚才说明的那样。无论您如何努力，都没有办法只使用应用层的代码以绝对保证在同一时刻只有一个线程访问 R。

对同步问题的解决方案简单而优雅。操作系统(在 Windows 中)提供了一个例程，在一个连续的操作中完成对 flag 的测试和设置(如果可能的话)。用操作系统工程师的话来说，这就是所谓的测试和置位(test and set)操作。由于历史的原因，用来控制对共享资源的访问并提供线程(以及进程)间同步的标记被称为信号量。信号量是 Windows 同步系统的核心。

2. Windows 的同步对象

Windows 支持几种类型的同步对象。第一种类型是经典的信号量。当使用信号量时，可以完全同步资源，在此情况下只有一个进程或者线程在同一时刻可以访问这个资源，或者信号量允许不超过一定数量的进程或者线程在同一时刻访问资源。信号量使用计数器来实现，当某个任务使用信号量时，计数器减小；当这个任务释放信号量时，计数器增加。

第二个同步对象是互斥体信号量，或者简称为互斥体。互斥体将一个资源同步，保证在任何时候都只有一个线程或者进程来访问它。在本质上，互斥体是标准信号量的一个特殊版本。

第三个同步对象是事件对象。它可以用来阻塞对某个资源的访问，直到某个其他的进程或者线程发送信号，通知可以使用资源(也就是一个事件对象发送某个指定的事件发生的信号)。

第四个同步对象是可等待计时器。可等待计时器阻塞线程的执行，直到指定的时间。也可以创建计时器序列，这是一个计时器的列表。

可以使用临界区对象将一个代码段放入临界区，从而阻止在同一时刻一个以上的线程使用这段代码。当一个线程进入临界区时，其他线程只有在第一个线程离开整个临界区时才可以使用它。

本章使用的惟一的同步对象是互斥体，下面的部分将对其进行描述。然而，C++程序员可以使用所有的 Windows 定义的同步对象。如前所述，这是使得 C++ 依赖于操作系统处理多线程的主要优点之一：所有的多线程特性都在您的控制之中。

3. 使用互斥体同步线程

如前所述，互斥体是一种特殊的信号量，在给定的时间内，只允许一个线程访问某个资源。在使用互斥体之前，必须使用 `CreatMutex()` 创建一个互斥体，函数原型如下：

```
HANDLE CreateMutex(LPSECURITY_ATTRIBUTES secAttr,  
                  BOOL acquire,  
                  LPCSTR name);
```

在此，`secAttr` 是用来描述安全属性的指针。如果 `secAttr` 为 `NULL`，则使用默认的安全描述符。

如果创建的线程需要互斥体的控制，则 `acquire` 为 `true`，否则为 `false`。

`name` 参数指向一个字符串，这个字符串是互斥体对象的名称。互斥体是一个全局对象，它可能被其他进程使用。为此，当两个进程都打开了使用相同名称的互斥体时，二者引用了相同的互斥体。使用这种方法可以将两个进程同步。这个名称也可以为 `NULL`，在此情况下这个信号量被限制在一个进程之内。

如果成功，则 `CreatMutex()` 函数返回信号量的句柄，否则，返回 `NULL`。当主进程结束时，互斥体的句柄则自动关闭。当不再需要时，可以调用 `CloseHandle()` 来显式地关闭互斥体的句柄。

当创建信号量时，可以调用两个相关的函数来使用它：`WaitForSingleObject()` 和 `ReleaseMutex()`。这两个函数的原型如下：

```
DWORD WaitForSingleObject(HANDLE hObject, DWORD howLong);  
BOOL ReleaseMutex(HANDLE hMutex);
```

`WaitForSingleObject()` 等待一个同步对象，直到这个对象可以使用或者超时之后才会返回。

在使用互斥体时, *hObject* 是互斥体的句柄。*howLong* 参数以毫秒为单位指定调用例程的等待时间。当这个时间用尽时, 会返回超时错误。为了无限期地等待, 可以使用值 *INFINITE*。当成功时(也就是访问被准许), 这个函数返回 *WAIT_OBJECT_0*。当发生超时时, 返回 *WAIT_TIMEOUT*。

ReleaseMutex() 释放互斥体, 并允许其他线程获取它。在此, *hMutex* 是互斥体的句柄。如果成功, 则函数返回非 0 值; 如果失败, 则返回 0。

为了使用互斥体控制对共享资源的访问, 封装了访问在调用 *WaitForSingleObject()* 和 *ReleaseMutex()* 之间的资源的代码, 如下面的代码所示(当然, 超时期限随应用程序的不同而不同)。

```
if(WaitForSingleObject(hMutex, 10000)==WAIT_TIMEOUT) {
    // handle time-out error
}

// access the resource

ReleaseMutex(hMutex);
```

通常, 您会选择足够长的超时期限来适应程序的操作。如果在开发多线程应用程序时重复出现超时错误, 那么通常意味着您创建了死锁条件。当一个线程等待另一个线程永远都不会释放的互斥体时, 就会发生死锁。

3.5 创建线程控制面板

当开发多线程程序时, 体验不同的优先级设置通常是有用的。能够动态地挂起或者重新启动线程, 甚至终止线程也是有用的。您将会看到, 使用刚才描述的线程函数来创建允许完成这些任务的线程控制面板相当容易。另外, 可以在您的多线程程序运行时使用控制面板。线程控制面板的动态特性使得您可以轻易地改变线程的执行配置文件并观察结果。

这个部分开发的线程控制面板能够控制一个线程。然而, 可以创建任意多个面板, 每个面板都可以控制不同的线程。为了简单起见, 线程控制面板用非模式对话框实现, 这个面板属于桌面而不是属于应用程序, 这个面板控制此应用程序的线程。

线程控制面板可以用来执行下面的操作:

- 设置线程的优先级
- 挂起线程
- 重新执行线程
- 终止线程

另外它还显示了线程的当前优先级设置。线程控制对话框如图 3-1 所示。

如前所述, 控制面板是非模式对话框。当非模式对话框激活时, 应用程序的其余部分仍然可以运行。因此, 控制面板的运行不依赖于它所使用的应用程序。

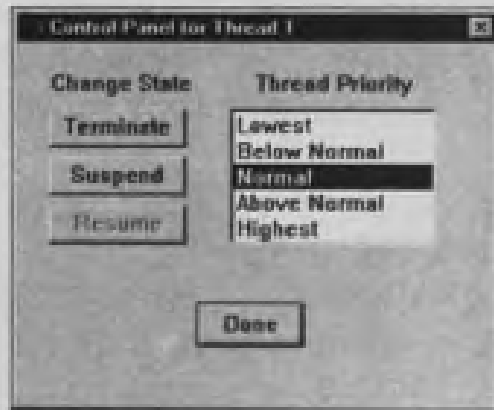


图 3-1 线程控制对话框

3.5.1 线程控制面板

线程控制面板的代码如下所示。文件名为 tcp.cpp。

```
// A thread control panel.

#include <map>
#include <windows.h>
#include "panel.h"
using namespace std;

const int NUMPRIORITIES = 5;
const int OFFSET = 2;

// Array of strings for priority list box.
char priorities[NUMPRIORITIES][80] = {
    "Lowest",
    "Below Normal",
    "Normal",
    "Above Normal",
    "Highest"
};

// A Thread Control Panel Class.
class ThrdCtrlPanel {
    // Information about the thread under control.
    struct ThreadInfo {
        HANDLE hThread; // handle of thread
        int priority;    // current priority
        bool suspended; // true if suspended
        ThreadInfo(HANDLE ht, int p, bool s) {
            hThread = ht;
            priority = p;
            suspended = s;
        }
    };

    // This map holds a ThreadInfo for each
```

```

// active thread control panel.
static map<HWND, ThreadInfo> dialogmap;

public:

// Construct a control panel.
ThrdCtrlPanel(HINSTANCE hInst, HANDLE hThrd);
// The control panel's callback function.
static LRESULT CALLBACK ThreadPanel(HWND hwnd, UINT message,
                                     WPARAM wParam, LPARAM lParam);
};

// Define static member dialogmap.
map<HWND, ThrdCtrlPanel::ThreadInfo>
ThrdCtrlPanel::dialogmap;

// Create a thread control panel.
ThrdCtrlPanel::ThrdCtrlPanel(HINSTANCE hInst,
                              HANDLE hThrd)
{
    ThreadInfo ti(hThrd,
                  GetThreadPriority(hThrd)+OFFSET,
                  false);

    // Owner window is desktop.
    HWND hDialog = CreateDialog(hInst, "ThreadPanelDB",
                                NULL,
                                (DLGPROC) ThreadPanel);

    // Put info about this dialog box in the map.
    dialogmap.insert(pair<HWND, ThreadInfo>(hDialog, ti));

    // Set the control panel's title.
    char str[80] = "Control Panel for Thread ";
    char str2[4];
    _itoa(dialogmap.size(), str2, 10);
    strcat(str, str2);
    SetWindowText(hDialog, str);

    // Offset each dialog box instance.
    MoveWindow(hDialog, 30*dialogmap.size(),
              30*dialogmap.size(),
              300, 250, 1);

    // Update priority setting in the list box.
    SendDlgItemMessage(hDialog, IDD_LB, LB_SETCURSEL,
                      (WPARAM) ti.priority, 0);

    // Increase priority to ensure control. You can
    // change or remove this statement based on your
    // execution environment.
    SetThreadPriority(GetCurrentThread(),

```

```

        THREAD_PRIORITY_ABOVE_NORMAL);
    }

// Thread control panel dialog box callback function.
LRESULT CALLBACK ThrdCtrlPanel::ThreadPanel(HWND hwnd,
                                             UINT message,
                                             WPARAM wParam,
                                             LPARAM lParam)
{
    int i;
    HWND hpbRes, hpbSus, hpbTerm;

    switch(message) {
        case WM_INITDIALOG:
            // Initialize priority list box.
            for(i=0; i<NUMPRIORITIES; i++) {
                SendDlgItemMessage(hwnd, IDD_LB,
                                    LB_ADDSTRING, 0, (LPARAM) priorities[i]);
            }

            // Set suspend and resume buttons for thread.
            hpbSus = GetDlgItem(hwnd, IDD_SUSPEND);
            hpbRes = GetDlgItem(hwnd, IDD_RESUME);
            EnableWindow(hpbSus, true); // enable Suspend
            EnableWindow(hpbRes, false); // disable Resume
            return 1;
        case WM_COMMAND:
            map<HWND, ThreadInfo>::iterator p = dialogmap.find(hwnd);

            switch(LOWORD(wParam)) {
                case IDD_TERMINATE:
                    TerminateThread(p->second.hThread, 0);

                    // Disable Terminate button.
                    hpbTerm = GetDlgItem(hwnd, IDD_TERMINATE);
                    EnableWindow(hpbTerm, false); // disable

                    // Disable Suspend and Resume buttons.
                    hpbSus = GetDlgItem(hwnd, IDD_SUSPEND);
                    hpbRes = GetDlgItem(hwnd, IDD_RESUME);
                    EnableWindow(hpbSus, false); // disable Suspend
                    EnableWindow(hpbRes, false); // disable Resume

                    return 1;
                case IDD_SUSPEND:
                    SuspendThread(p->second.hThread);

                    // Set state of the Suspend and Resume buttons.
                    hpbSus = GetDlgItem(hwnd, IDD_SUSPEND);
                    hpbRes = GetDlgItem(hwnd, IDD_RESUME);
                    EnableWindow(hpbSus, false); // disable Suspend

```

```

        EnableWindow(hpbRes, true); // enable Resume

        p->second.suspended = true;
        return 1;
    case IDD_RESUME:
        ResumeThread(p->second.hThread);

        // Set state of the Suspend and Resume buttons.
        hpbSus = GetDlgItem(hwnd, IDD_SUSPEND);
        hpbRes = GetDlgItem(hwnd, IDD_RESUME);
        EnableWindow(hpbSus, true); // enable Suspend
        EnableWindow(hpbRes, false); // disable Resume

        p->second.suspended = false;
        return 1;
    case IDD_LB:
        // If a list box entry was clicked,
        // then change the priority.
        if (HIWORD(wParam) == LBN_DBLCLK) {
            p->second.priority = SendDlgItemMessage(hwnd,
                IDD_LB, LB_GETCURSEL,
                0, 0);
            SetThreadPriority(p->second.hThread,
                p->second.priority - OFFSET);
        }
        return 1;
    case IDCANCEL:
        // If thread is suspended when panel is closed,
        // then resume thread to prevent deadlock.
        if (p->second.suspended) {
            ResumeThread(p->second.hThread);
            p->second.suspended = false;
        }

        // Remove this thread from the list.
        dialogmap.erase(hwnd);

        // Close the panel.
        DestroyWindow(hwnd);
        return 1;
    }
}
return 0;
}

```

控制面板需要如下的资源文件，称为 tcp.rc。

```

#include <windows.h>
#include "panel.h"

ThreadPanelDB DIALOGEX 20, 20, 140, 110
CAPTION "Thread Control Panel"

```



```

STYLE WS_BORDER | WS_VISIBLE | WS_POPUP | WS_CAPTION | WS_SYSMENU
{
    DEFPUSHBUTTON "Done", IDCANCEL, 55, 80, 33, 14
    PUSHBUTTON "Terminate", IDD_TERMINATE, 10, 20, 42, 12
    PUSHBUTTON "Suspend", IDD_SUSPEND, 10, 35, 42, 12
    PUSHBUTTON "Resume", IDD_RESUME, 10, 50, 42, 12
    LISTBOX IDD_LB, 65, 20, 63, 42, LBS_NOTIFY | WS_VISIBLE |
        WS_BORDER | WS_VSCROLL | WS_TABSTOP
    CTEXT "Thread Priority", IDD_TEXT1, 65, 8, 64, 10
    CTEXT "Change State", IDD_TEXT2, 0, 8, 64, 10
}

```

控制面板使用了如下的头文件 `panel.h`

```

#define IDD_LB          200
#define IDD_TERMINATE 202
#define IDD_SUSPEND    204
#define IDD_RESUME     206
#define IDD_TEXT1      208
#define IDD_TEXT2      209

```

为了使用线程控制面板，需要遵循如下步骤：

- (1) 在程序中包含 `tcp.cpp`
- (2) 在程序的资源文件中包含 `tcp.rc`
- (3) 创建想要控制的一个线程或多个线程
- (4) 为每个线程实例化 `ThrdCtrlPanel` 对象

每个 `ThrdCtrlPanel` 对象链接线程和控制这个线程的对话框。

对于多个文件需要访问 `ThrdCtrlPanel` 的大项目，可以使用头文件 `tcp.h` 来包含 `ThrdCtrlPanel` 的声明。`tcp.h` 的内容如下：

```

// A header file for the ThrdCtrlPanel class.

class ThrdCtrlPanel {
public:

    // Construct a control panel.
    ThrdCtrlPanel(HINSTANCE hInst, HANDLE hThrd);

    // The control panel's callback function.
    static LRESULT CALLBACK ThreadPanel(HWND hwnd, UINT message,
        WPARAM wParam, LPARAM lParam);
};

```

3.5.2 线程控制面板的详细分析

让我们仔细观察这个线程控制面板。开始它进行了如下的全局定义：

```

const int NUMPRIORITIES = 5;
const int OFFSET = 2;

// Array of strings for priority list box.

```

```
char priorities[NUMPRIORITIES][80] = {
    "Lowest",
    "Below Normal",
    "Normal",
    "Above Normal",
    "Highest"
};
```

`priorities` 数组包含了对应于线程的优先级设置的字符串。它初始化控制面板内的列表框，这个列表框显示了当前线程优先级。优先级的数量由 `NUMPRIORITIES` 指定，对于 Windows 这个值为 5。因此，`NUMPRIORITIES` 定义了线程可能具有的不同优先级的数量。(如果您在另一个操作系统中使用了这些代码，则可能需要其他的值)。通过使用控制面板，可以为线程设置如下的优先级：

```
THREAD_PRIORITY_HIGHEST
THREAD_PRIORITY_ABOVE_NORMAL
THREAD_PRIORITY_NORMAL
THREAD_PRIORITY_BELOW_NORMAL
THREAD_PRIORITY_LOWEST
```

另外两个线程优先级的设置：

```
THREAD_PRIORITY_TIME_CRITICAL
THREAD_PRIORITY_IDLE
```

没有被支持，是因为对于这个控制面板，它们没有实际的意义。例如，如果想要创建一个 time-critical 应用程序，最好将它的优先级类别设置为 time-critical。

`OFFSET` 定义了列表框索引和线程优先级之间转换的偏移量。应该记得普通优先级的值为 0。在此示例中，最高的优先级为 `THREAD_PRIORITY_HIGHEST`，其值为 2。最低的优先级为 `THREAD_PRIORITY_LOWEST`，其值为 -2。由于列表框索引以 0 开始，因此使用了这个偏移量来在索引和优先级设置之间转换。

随后声明了 `ThrdCtrlPanel` 类。下面给出类的声明代码：

```
// A Thread Control Panel Class.
class ThrdCtrlPanel {
    // Information about the thread under control.
    struct ThreadInfo {
        HANDLE hThread; // handle of thread
        int priority;    // current priority
        bool suspended; // true if suspended
        ThreadInfo(HANDLE ht, int p, bool s) {
            hThread = ht;
            priority = p;
            suspended = s;
        }
    };

    // This map holds a ThreadInfo for each
    // active thread control panel.
    static map<HWND, ThreadInfo> dialogmap;
```

关于被控制线程的信息保存在 `ThreadInfo` 类型的结构中。这个线程的句柄保存在 `hThread` 中。它的优先级保存在 `priority` 中。如果这个线程被挂起, `suspended` 则为 `true`, 否则为 `false`。

静态成员 `dialogmap` 是一个 STL `map` 对象, 链接线程信息与用来控制线程的对话框的句柄。由于在给定的时间可以具有多个活动的线程控制面板, 因此必须有某种方法来判断某个线程与哪个面板相关。`dialogmap` 提供了这个链接。

1. `ThreadCtrlPanel` 的构造函数

`ThreadCtrlPanel` 的构造函数如下所示。应用程序的实例句柄和被控制的线程的句柄传递给这个构造函数。实例句柄用来创建控制面板对话框。

```
// Create a thread control panel.
ThrdCtrlPanel::ThrdCtrlPanel(HINSTANCE hInst,
                              HANDLE hThrd)
{
    ThreadInfo ti(hThrd,
                  GetThreadPriority(hThrd)+OFFSET,
                  false);

    // Owner window is desktop.
    HWND hDialog = CreateDialog(hInst, "ThreadPanelDB",
                                NULL,
                                (DLGPROC) ThreadPanel);

    // Put info about this dialog box in the map.
    dialogmap.insert(pair<HWND, ThreadInfo>(hDialog, ti));

    // Set the control panel's title.
    char str[80] = "Control Panel for Thread ";
    char str2[4];
    _itoa(dialogmap.size(), str2, 10);
    strcat(str, str2);
    SetWindowText(hDialog, str);

    // Offset each dialog box instance.
    MoveWindow(hDialog, 30*dialogmap.size(),
              30*dialogmap.size(),
              300, 250, 1);

    // Update priority setting in the list box.
    SendDlgItemMessage(hDialog, IDD_LB, LB_SETCURSEL,
                      (WPARAM) ti.priority, 0);

    // Increase priority to ensure control. You can
    // change or remove this statement based on your
    // execution environment.
    SetThreadPriority(GetCurrentThread(),
                     THREAD_PRIORITY_ABOVE_NORMAL);
}
```

在这个构造函数的开始, 创建了一个 `ThreadInfo` 实例 `ti`, `ti` 包含了线程的初始设置。注意, 这个优先级是通过为被控制的线程调用 `GetThreadPriority()` 获得的。随后, 通过调用 `CreateDialog()` 创建控制面板对话框。`CreateDialog()` 是用来创建非模式对话框的 Windows API 函数, 使得对话框与创建它的应用程序无关。这个对话框的句柄被返回并存储在 `hDialog` 中。然后, `hDialog` 和包含在 `ti` 中的线程信息保存在 `dialogmap` 中。从而线程与控制它的对话框链接起来。

随后设置对话框的标题以表示线程的数量。线程数量的获取基于 `dialogmap` 中条目的数量。您可能想要实现另一种方法, 显式地将每个线程的名称传递给 `ThrdCtrlPanel` 的构造函数。然而对于本章而言, 只要有线程的数量就足够了。

随后, 通过调用另一个 Windows API 函数 `MoveWindow()`, 控制面板在屏幕上的位置就可以移动。这使得被显示的多个面板之间不会出现遮挡的情况。然后通过调用 Windows API 函数 `SendDlgItemMessage()`, 线程的优先级设置就显示在优先级列表框中。

最后, 当前线程的优先级增加到高于普通值。从而确保了应用程序接收足够的 CPU 时间来响应用户的输入, 无论被控制线程的优先级如何。并不是所有的情况都需要这个步骤, 您可以尝试它。

2. `ThreadPanel()` 函数

`ThreadPanel()` 是 Windows 的回调函数, 用来响应用户与线程控制面板的交互。像所有的对话框回调函数一样, 每当用户改变了控制状态时, 它就会收到一个消息。它传递发生操作的那个对话框的句柄、消息以及这条消息所需的任何附加信息。它的常用操作模式与对话框使用的其他回调函数相同。下面的讨论描述了每条消息发生的事件。

在线程控制面板对话框首次创建时, 它接收到 `WM_INITDIALOG` 消息, 由下面的 case 序列处理:

```
case WM_INITDIALOG:
    // Initialize priority list box.
    for(i=0; i<NUMPRIORITIES; i++) {
        SendDlgItemMessage(hwnd, IDD_LB,
            LB_ADDSTRING, 0, (LPARAM) priorities[i]);
    }

    // Set Suspend and Resume buttons for thread.
    hpbSus = GetDlgItem(hwnd, IDD_SUSPEND);
    hpbRes = GetDlgItem(hwnd, IDD_RESUME);
    EnableWindow(hpbSus, true); // enable Suspend
    EnableWindow(hpbRes, false); // disable Resume
    return 1;
```

这段代码初始化了优先级列表框, 并为初始状态设置了 `Suspend` 和 `Resume` 按钮, `Suspend` 按钮可用, `Resume` 按钮不可用。

用户的每一个交互都会产生 `WM_COMMAND` 消息。每次接收到这个消息时, 获取指向 `dialogmap` 中这个对话框条目的迭代器, 如下所示:

```
case WM_COMMAND:
    map<HWND, ThreadInfo>::iterator p = dialogmap.find(hwnd);
```

使用 `p` 指向的信息恰当地处理每个操作。由于 `p` 是一个 `map` 对象的迭代器，它指向一个 `pair` 类型的对象，`pair` 是 STL 定义的一个结构。这个结构包含了两个字段：`first` 和 `second`。这两个字段分别对应于由键和值组成的信息。在此情况下，句柄为键，线程信息为值。

精确指示所发生的操作的代码包含在 `wParam` 的低位字中，用它来控制处理剩余消息的 `switch` 语句。下面将逐一讲述。

当用户按下 `Terminate` 按钮时，终止受控的线程。这由下面的 `case` 序列来处理：

```
case IDD_TERMINATE:
    TerminateThread(p->second.hThread, 0);

    // Disable Terminate button.
    hpbTerm = GetDlgItem(hwnd, IDD_TERMINATE);
    EnableWindow(hpbTerm, false); // disable

    // Disable Suspend and Resume buttons.
    hpbSus = GetDlgItem(hwnd, IDD_SUSPEND);
    hpbRes = GetDlgItem(hwnd, IDD_RESUME);
    EnableWindow(hpbSus, false); // disable Suspend
    EnableWindow(hpbRes, false); // disable Resume

    return 1;
```

通过调用 `TerminateThread()` 来终止线程。注意线程句柄的获取方式。如前所述，由于 `p` 是 `map` 对象的一个迭代器，因此它指向 `pair` 类型的一个对象，`pair` 包含了在 `first` 字段中的键以及 `second` 字段中的值。这就是可以使用表达式 `p->second.hThread` 获取线程句柄的原因。该线程停止后，`Terminate` 按钮将失效。

一旦线程终止，将不可恢复。注意控制面板使用 `TerminateThread()` 来终止线程的执行。如前所述，必须小心使用这个函数。如果您使用控制面板来试验您自己的线程，就要确保不会出现有害的影响。

当用户按下 `Suspend` 按钮时，挂起线程。下面的序列完成了这个操作：

```
case IDD_SUSPEND:
    SuspendThread(p->second.hThread);

    // Set state of the Suspend and Resume buttons.
    hpbSus = GetDlgItem(hwnd, IDD_SUSPEND);
    hpbRes = GetDlgItem(hwnd, IDD_RESUME);
    EnableWindow(hpbSus, false); // disable Suspend
    EnableWindow(hpbRes, true);  // enable Resume

    p->second.suspended = true;
    return 1;
```

通过调用 `SuspendThread()` 来挂起线程。随后，`Suspend` 和 `Resume` 按钮的状态被更新，`Resume` 按钮可以使用，`Suspend` 按钮不可使用。从而避免了用户试图将一个线程挂起两次。

当按下 `Resume` 按钮时，恢复挂起的线程。由下面的代码来处理：

```
case IDD_RESUME:
```

```
ResumeThread(p->second.hThread);
```

```
// Set state of the Suspend and Resume buttons.  
hpbSus = GetDlgItem(hwnd, IDD_SUSPEND);  
hpbRes = GetDlgItem(hwnd, IDD_RESUME);  
EnableWindow(hpbSus, true); // enable Suspend  
EnableWindow(hpbRes, false); // disable Resume
```

```
p->second.suspended = false;  
return 1;
```

通过调用 `ResumeThread()` 来恢复线程，恰当地设置 `Suspend` 和 `Resume` 按钮。

为了改变线程的优先级，用户可以双击优先级列表框中的条目。对这个事件的处理如下所示：

```
case IDD_LB:  
    // If a list box entry was double-clicked,  
    // then change the priority.  
    if(HIWORD(wParam)==LBN_DBLCLK) {  
        p->second.priority = SendDlgItemMessage(hwnd,  
            IDD_LB, LB_GETCURSEL,  
            0, 0);  
  
        SetThreadPriority(p->second.hThread,  
            p->second.priority-OFFSET);  
    }  
    return 1;
```

列表框生成了不同类型的提示消息来描述所发生事件的精确类型。提示消息包含在 `wParam` 的高位字中。这些消息的其中之一是 `LBN_DBLCLK`，这意味着用户双击了列表框中的一个条目。当接收到这个提示消息时，通过调用 Windows API 函数 `SendDlgItemMessage()` 来获取这个条目的索引，来请求当前的选择。然后使用这个值来设置线程的优先级。注意减去 `OFFSET` 来规范索引的值。

最后，当用户关闭线程控制面板对话框时，会发送 `IDCANCEL` 消息。它由如下序列处理：

```
case IDCANCEL:  
    // If thread is suspended when panel is closed,  
    // then resume thread to prevent deadlock.  
    if(p->second.suspended) {  
        ResumeThread(p->second.hThread);  
        p->second.suspended = false;  
    }  
  
    // Remove this thread from the list.  
    dialogmap.erase(hwnd);  
  
    // Close the panel.  
    DestroyWindow(hwnd);  
    return 1;
```

如果挂起线程，则会重新开始线程。为了防止不小心死锁线程，这样做是有必要的。随后，

删除 dialogmap 中这个对话框的条目。最后,调用 Windows API 函数 DestroyWindow()来删除此对话框。

3.5.3 控制面板的演示

在此有一个包含了线程控制面板的程序来说明它的使用。示例输出在图 3-2 中显示。程序创建了一个主窗口并定义两个子线程。当开始的时候,这些线程简单地从 0 到 50000 计数,并在主窗口中显示这个计数。这些线程可以通过激活一个线程控制面板来控制。

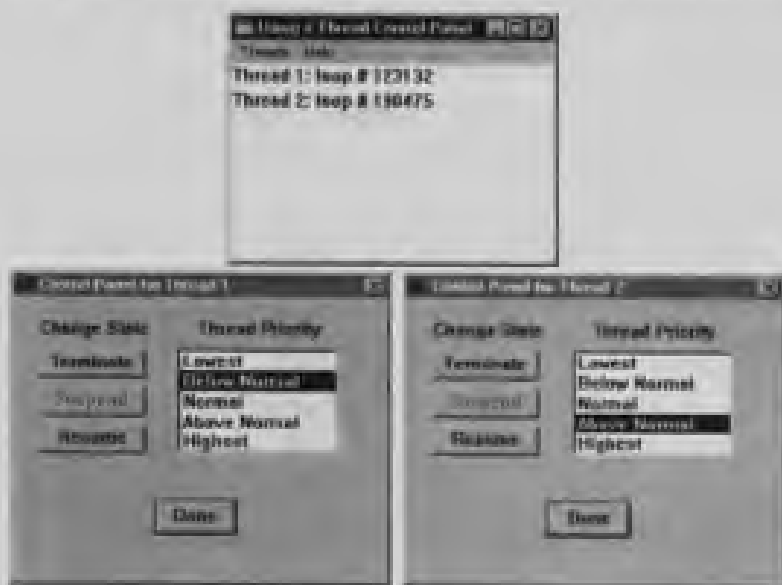


图 3-2 线程控制面板示例程序的输出

为了使用这个程序,首先在 Threads 菜单中选择 Start Threads(或者按 F2)来开始执行线程,然后选择 Threads 菜单中的 Control Panels(或者按 F3)来激活线程控制面板。当控制面板激活时,您可以设置不同的优先级。

提示:

对 Windows 程序设计的讲述超出了本书的范围。然而,示例程序的操作相当的直接,所有的 Windows 程序员应该都可以很容易地理解它们。

```
// Demonstrate the thread control panel.
#include <windows.h>
#include <process.h>
#include "thrdapp.h"
#include "tcp.cpp"
```

```
const int MAX = 50000;
```

```
LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);
```

```
unsigned __stdcall MyThread1(void * param);
```

```
unsigned __stdcall MyThread2(void * param);
```

```
char str[255]; // holds output strings
```

```

unsigned tid1, tid2; // thread IDs
HANDLE hThread1, hThread2; // thread handles

HINSTANCE hInst; // instance handle

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                  LPSTR args, int winMode)
{
    HWND hwnd;
    MSG msg;
    WNDCLASSEX wcl;
    HACCEL hAccel;

    // Define a window class.
    wcl.cbSize = sizeof(WNDCLASSEX);

    wcl.hInstance = hThisInst; // handle to this instance
    wcl.lpszClassName = "MyWin"; // window class name
    wcl.lpfnWndProc = WindowFunc; // window function
    wcl.style = 0; // default style

    wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); // large icon
    wcl.hIconSm = NULL; // use small version of large icon
    wcl.hCursor = LoadCursor(NULL, IDC_ARROW); // cursor style
    wcl.lpszMenuName = "ThreadAppMenu"; // main menu

    wcl.cbClsExtra = 0; // no extra memory needed
    wcl.cbWndExtra = 0;

    // Make the window background white.
    wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

    // Register the window class.
    if(!RegisterClassEx(&wcl)) return 0;

    /* Now that a window class has been registered, a window
       can be created. */
    hwnd = CreateWindow(
        wcl.lpszClassName, // name of window class
        "Using a Thread Control Panel", // title
        WS_OVERLAPPEDWINDOW, // window style - normal
        CW_USEDEFAULT, // X coordinate - let Windows decide
        CW_USEDEFAULT, // Y coordinate - let Windows decide
        260,           // width
        200,           // height
        NULL,          // no parent window
        NULL,          // no override of class menu
        hThisInst,     // instance handle
        NULL           // no additional arguments
    );
}

```



```

hInst = hThisInst; // save instance handle

// Load the keyboard accelerators.
hAccel = LoadAccelerators(hThisInst, "ThreadAppMenu");

// Display the window.
ShowWindow(hwnd, winMode);
UpdateWindow(hwnd);

// Create the message loop.
while(GetMessage(&msg, NULL, 0, 0))
{
    if(!TranslateAccelerator(hwnd, hAccel, &msg)) {
        TranslateMessage(&msg); // translate keyboard messages
        DispatchMessage(&msg); // return control to Windows
    }
}
return msg.wParam;
}

/* This function is called by Windows and is passed
   messages from the message queue.
*/
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                           WPARAM wParam, LPARAM lParam)
{
    int response;

    switch(message) {
        case WM_COMMAND:
            switch(LOWORD(wParam)) {
                case IDM_THREAD: // create the threads
                    hThread1 = (HANDLE) _beginthreadex(NULL, 0,
                                                         MyThread1, (void *) hwnd,
                                                         0, &tid1);
                    hThread2 = (HANDLE) _beginthreadex(NULL, 0,
                                                         MyThread2, (void *) hwnd,
                                                         0, &tid2);

                    break;
                case IDM_PANEL: // activate control panel
                    ThrdCtrlPanel(hInst, hThread1);
                    ThrdCtrlPanel(hInst, hThread2);
                    break;
                case IDM_EXIT:
                    response = MessageBox(hwnd, "Quit the Program?",
                                           "Exit", MB_YESNO);
                    if(response == IDYES) PostQuitMessage(0);
                    break;
                case IDM_HELP:
                    MessageBox(hwnd,
                               "F1: Help\nF2: Start Threads\nF3: Panel",
                               "Help", MB_OK);
            }
    }
}

```

```

        break;
    }
    break;
case WM_DESTROY: // terminate the program
    PostQuitMessage(0);
    break;
default:
    return DefWindowProc(hwnd, message, wParam, lParam);
}
return 0;
}

```

// First thread.

```

unsigned __stdcall MyThread1(void * param)
{
    int i;
    HDC hdc;

    for(i=0; i<MAX; i++) {
        wsprintf(str, "Thread 1: loop # %5d ", i);
        hdc = GetDC((HWND) param);
        TextOut(hdc, 1, 1, str, lstrlen(str));
        ReleaseDC((HWND) param, hdc);
    }

    return 0;
}

```

// Second thread.

```

unsigned __stdcall MyThread2(void * param)
{
    int i;
    HDC hdc;

    for(i=0; i<MAX; i++) {
        wsprintf(str, "Thread 2: loop # %5d ", i);
        hdc = GetDC((HWND) param);
        TextOut(hdc, 1, 20, str, lstrlen(str));
        ReleaseDC((HWND) param, hdc);
    }

    return 0;
}

```

此程序需要的头文件 `thrdapp.h` 如下所示:

```

#define IDM_THREAD 100
#define IDM_HELP 101
#define IDM_PANEL 102
#define IDM_EXIT 103

```

程序需要的资源文件如下所示:

```
#include <windows.h>
#include "thrdapp.h"
#include "tcp.rc"

ThreadAppMenu MENU
{
    POPUP "&Threads" {
        MENUITEM "&Start Threads\tF2", IDM_THREAD
        MENUITEM "&Control Panels\tF3", IDM_PANEL
        MENUITEM "E&xit\tCtrl+X", IDM_EXIT
    }
    MENUITEM "&Help", IDM_HELP
}

ThreadAppMenu ACCELERATORS
{
    VK_F1, IDM_HELP, VIRTKEY
    VK_F2, IDM_THREAD, VIRTKEY
    VK_F3, IDM_PANEL, VIRTKEY
    "^X", IDM_EXIT
}
```

3.6 一个多线程的垃圾回收器

在开发多线程程序时, 尽管通过线程控制面板来控制线程是有用的, 然而最终是使用线程以使得它们重要。为此, 本章给出了第2章开发的原始的 GCPtr 垃圾回收器类的多线程版本。第2章显示的 GCPtr 的版本在 GCPtr 对象超出作用域时回收不再使用的内存。尽管这种方法适合于某些应用程序, 然而通常更好的选择是使得垃圾回收器作为后台任务运行, 在 CPU 循环空闲的时候回收内存。这里开发的产品是为 Windows 设计的, 但是相似的基本技术可以应用到其他的多线程环境。

将 GCPtr 转换为后台任务相当容易, 但是会涉及到一些改变。主要有以下几点:

(1) 支持线程的成员变量必须加入到 GCPtr 中。这些变量包括线程句柄、互斥体句柄以及跟踪现有的 GCPtr 对象数量的实例计数器。

(2) GCPtr 的构造函数必须开始垃圾回收线程。另外构造函数还必须创建控制同步的互斥体。这只能在创建第一个 GCPtr 对象时发生一次。

(3) 必须定义另外的异常来指示超时情况。

(4) GCPtr 析构函数不应该再调用 collect()。垃圾回收由垃圾回收线程来处理。

(5) 必须为垃圾回收器定义一个名为 gc() 的函数作为线程的入口点。

(6) 必须定义一个函数 isRunning(), 如果正在使用垃圾回收, 它就返回 true。

(7) 访问包含在 gclist 中的垃圾回收链表的 GCPtr 的成员函数必须被同步, 从而在某个时刻只有一个线程可以访问这个链表。

下面的部分给出了这些改变。

3.6.1 附加的成员变量

GCPtr 的多线程版本要求加入如下的成员变量:

```
// These support multithreading.
unsigned tid; // thread id
static HANDLE hThrd; // thread handle
static HANDLE hMutex; // handle of mutex

static int instCount; // counter of GCPtr objects
```

垃圾回收器使用的线程的 ID 存储在 tid 中。只有在调用_beginthreadex()时才会使用这个成员。这个线程的句柄存储在 hThrd 中。互斥体用来同步访问 GCPtr 的句柄存储在 hMutex 中。当前 GCPtr 对象的数量保存在 instCount 中。最后 3 个变量为静态变量, 它们被 GCPtr 的所有实例共享。它们在 GCPtr 的外部定义如下:

```
template <class T, int size>
int GCPtr<T, size>::instCount = 0;

template <class T, int size>
HANDLE GCPtr<T, size>::hMutex = 0;

template <class T, int size>
HANDLE GCPtr<T, size>::hThrd = 0;
```

3.6.2 多线程的 GCPtr 构造函数

除了原先的任务之外, 多线程的 GCPtr()必须创建互斥体, 开始垃圾回收器线程, 并更新实例计数器。在此是更新了的版本:

```
// Construct both initialized and uninitialized objects.
GCPtr(T *t=NULL) {

    // When first object is created, create the mutex
    // and register shutdown().
    if(hMutex==0) {
        hMutex = CreateMutex(NULL, 0, NULL);
        atexit(shutdown);
    }

    if(WaitForSingleObject(hMutex, 10000)==WAIT_TIMEOUT)
        throw TimeOutExc();

    list<GCInfo<T> >::iterator p;

    p = findPtrInfo(t);

    // If t is already in gclist, then
    // increment its reference count.
    // Otherwise, add it to the list.
    if(p != gclist.end())
```

```

        p->refcount++; // increment ref count
    else {
        // Create and store this entry.
        GCInfo<T> gcObj(t, size);
        gclist.push_front(gcObj);
    }

    addr = t;
    arraySize = size;
    if(size > 0) isArray = true;
    else isArray = false;

    // Increment instance counter for each new object.
    instCount++;

    // If the garbage collection thread is not
    // currently running, start it running.
    if(hThrd==0) {
        hThrd = (HANDLE) _beginthreadex(NULL, 0, gc,
            (void *) 0, 0, (unsigned *) &tid);

        // For some applications, it will be better
        // to lower the priority of the garbage collector
        // as shown here:
        //
        // SetThreadPriority(hThrd,
        //                     THREAD_PRIORITY_BELOW_NORMAL);
    }

    ReleaseMutex(hMutex);
}

```

让我们仔细分析这段代码。首先，如果 `hMutex` 为 0，则意味着将要创建第一个 `GCPtr`，并且还没有为垃圾回收器创建互斥体。如果是这种情况，则创建互斥体，将其句柄赋给 `hMutex`。同时，通过调用 `atexit()`，将函数 `shutdown()` 注册为终止函数。

注意，在多线程的垃圾回收器中，`shutdown()` 有两个作用。首先，如同原先版本的 `GCPtr`，`shutdown()` 释放任何不再使用的但是由于循环引用而没有被释放的内存。其次，当使用了多线程的垃圾回收器结束的时候，它终止了垃圾回收线程。这意味着仍然存在没有被释放的动态分配的内存。这很重要，因为这些对象可能具有需要被调用的析构函数。由于 `shutdown()` 释放所有仍然存在的对象，因此它也会释放这些对象。

随后，通过调用 `WaitForSingleObject()` 来获取互斥体。为了阻止两个线程在同一时刻访问 `gclist`，这是必须的。一旦获取了互斥体，就开始搜索 `gclist`，查找是否存在与 `t` 中的地址匹配的条目。如果能够找到，其引用计数增加。如果没有先前存在的条目与 `t` 匹配，就创建新的 `GCInfo` 对象来包含这个地址，并且将这个对象加入到 `gclist` 中。然后设置 `addr`、`arraySize` 和 `isArray`。这些操作与先前的 `GCPtr` 的版本相同。

然后 `instCount` 递增。记得 `instCount` 是初始化为 0 的。每当有对象创建的时候，都会递增以跟踪现有的 `GCPtr` 对象的数目。只要这个计数大于 0，垃圾回收器就会持续运行。

随后, 如果 `hThrd` 为 0(如同它的初始化状态), 则说明还没有为垃圾回收器创建线程。在此情况下, 调用 `_beginthreadex()` 来开始这个线程。这个线程的句柄被赋给 `hThrd`。这个线程的入口函数称为 `gc()`, 对它会进行简短的检查。

最后, 释放互斥体, 返回构造函数。有必要指出, 每一个对 `WaitForSingleObject()` 的调用都必须通过调用 `ReleaseMutex()` 来平衡, 在 `GCPtr` 的构造函数中显示了这一点。释放互斥体失败会引发死锁。

3.6.3 TimeOutExc 异常

在前面部分对 `GCPtr()` 代码的描述中, 您可能已经发现, 当 10 秒钟之后仍然不能获取互斥体时, 就会抛出 `TimeOutExc` 异常。坦白的说, 10 秒种是一个非常长的时间, 因此除非操作系统的任务调度程序被破坏, 否则不会发生超时。然而, 在确实发生超时的情况下, 您的应用程序代码可能想要捕捉这个异常。`TimeOutExc` 类如下所示:

```
// Exception thrown when a time-out occurs
// when waiting for access to hMutex.
//
class TimeOutExc {
    // Add functionality if needed by your application.
};
```

注意, 它没有包含数字。在本章, 它作为一个独特的类型存在就足够了。当然, 如果需要, 可以添加功能。

3.6.4 多线程的 GCPtr 析构函数

不同于单线程版本的 `GCPtr` 的析构函数, `~GCPtr()` 的多线程版本没有调用 `collect()`。取而代之的是, 它只是简单地递减超出作用域的 `GCPtr` 所指向内存的引用计数。实际的垃圾回收(如果存在的话)由垃圾回收线程处理。析构函数同时还递减实例计数器 `instCount` 的值。

多线程版本的 `~GCPtr()` 如下所示:

```
// Destructor for GCPtr.
template <class T, int size>
GCPtr<T, size>::~~GCPtr() {
    if(WaitForSingleObject(hMutex, 10000) == WAIT_TIMEOUT)
        throw TimeOutExc();

    list<GCInfo<T> >::iterator p;

    p = findPtrInfo(addr);
    if(p->refcount) p->refcount--; // decrement ref count

    // Decrement instance counter for each object
    // that is destroyed.
    instCount--;

    ReleaseMutex(hMutex);
}
```

3.6.5 gc()函数

垃圾回收器的入口函数称为 gc(), 如下所示:

```
// Entry point for garbage collector thread.
template <class T, int size>
unsigned __stdcall GCPtr<T, size>::gc(void * param) {
    #ifdef DISPLAY
        cout << "Garbage collection started.\n";
    #endif

    while(isRunning()) {
        collect();
    }

    collect(); // collect garbage on way out

    // Release and reset the thread handle so
    // that the garbage collection thread can
    // be restarted if necessary.
    CloseHandle(hThrd);
    hThrd = 0;

    #ifdef DISPLAY
        cout << "Garbage collection terminated for "
              << typeid(T).name() << "\n";
    #endif

    return 0;
}
```

gc()函数非常简单: 只要使用垃圾回收器它就会运行。如果 instCount 大于 0, isRunning() 函数就返回 true(这意味着仍然需要垃圾回收器), 否则, 就返回 false。在循环内连续调用 collect()。为了说明多线程垃圾回收器的运行, 这种方法是恰当的, 但是对于实际的应用, 它可能效率太低了。您可能想对 collect()的调用不那么频繁, 例如只有在内存变少时才调用它。您还可以在每次调用 collect()之后调用 Windows API 函数 Sleep()。Sleep()将调用的线程暂停执行指定的毫秒数。当睡眠时, 线程不会消耗 CPU 时间。

当 isRunning()返回值为 false 时, 循环结束, 导致 gc()最终结束, 从而终止了垃圾回收线程。由于多线程, 虽然 isRunning()返回了 false, 但是在 gclist 中可能仍然存在没有被释放的条目。为了处理这种情况, 在 gc()结束之前, 最后一次调用 collect()。

最后, 通过调用 Windows API 函数 CloseHandle()释放了线程句柄, 其值被设置为 0。将 hThrd 设置为 0 可以使得后来在程序中创建新的 GCPtr 对象时, GCPtr 构造函数重新启动这个线程。

3.6.6 isRunning()函数

isRunning()函数如下所示:

```
// Returns true if the collector is still in use.
static bool isRunning() { return instCount > 0; }
```

它只是简单地将 instCount 与 0 进行比较。只要 instCount 大于 0, 则当前至少存在一个 GCPtr 指针, 仍然需要垃圾回收器。

3.6.7 gclist 的同步访问

GCPtr 中的许多函数访问了 gclist, gclist 拥有垃圾回收链表。对 gclist 的访问必须同步, 以阻止在同一时刻两个或者多个线程试图使用它。原因很容易理解。如果访问没有同步, 那么某个线程可能从这个链表的结尾获取了一个迭代器, 同时另一个线程增加或者删除了这个链表的元素。在此情况下, 这个迭代器变得无效。为了阻止这个问题, 互斥体必须准许每一个访问 gclist 的代码序列。在此给出了 GCPtr 的复制构造函数的一个示例:

```
// Copy constructor.
GCPtr(const GCPtr &ob) {
    if(WaitForSingleObject(hMutex, 10000)==WAIT_TIMEOUT)
        throw TimeOutExc();

    list<GCInfo<T> >::iterator p;

    p = findPtrInfo(ob.addr);
    p->refcount++; // increment ref count

    addr = ob.addr;
    arraySize = ob.arraySize;
    if(arraySize > 0) isArray = true;
    else isArray = false;

    instCount++; // increase instance count for copy

    ReleaseMutex(hMutex);
}
```

注意复制构造函数做的第一件事是获取互斥体。然后创建这个对象的副本, 并调整指向内存的引用计数。当它结束时, 复制构造函数释放互斥体。相同的基本方法适用于所有访问 gclist 的函数。

3.6.8 其他两个改变

必须对原先的垃圾回收器做另外两个修改。首先, 原始版本的 GCPtr 定义了静态变量 first, 来指示创建第一个 GCPtr 的时刻。由于 hMutex 执行了这个功能, 因此这个变量不再需要, 从而从 GCPtr 中删除 first。因为它是一个静态变量, 所以还需要在 GCPtr 外部删除它的定义。

在原先单线程版本的垃圾回收器中, 如果您定义了 DISPLAY 宏, 就可以观察垃圾回收器的运转状态。在多线程版本中, 这些代码的大部分都被删除了, 因为在大多数情况下, 多线程使得输出混乱而难以理解。对于多线程的版本, 定义 DISPLAY 宏只能让您知道垃圾回收器的开始和结束时间。

3.6.9 完整的多线程垃圾回收器

多线程垃圾回收器的完整版本如下所示。这个文件称为 gcthrd.h。

```
// A garbage collector that runs as a background task.

#include <iostream>
```



```

#include <list>
#include <typeinfo>
#include <cstdlib>
#include <windows.h>
#include <process.h>

using namespace std;

// To watch the action of the garbage collector, define DISPLAY.
// #define DISPLAY

// Exception thrown when an attempt is made to
// use an Iter that exceeds the range of the
// underlying object.
//
class OutOfRangeExc {
    // Add functionality if needed by your application.
};

// Exception thrown when a time-out occurs
// when waiting for access to hMutex.
//
class TimeOutExc {
    // Add functionality if needed by your application.
};

// An iterator-like class for cycling through arrays
// that are pointed to by GCPtrs. Iter pointers
// ** do not ** participate in or affect garbage
// collection. Thus, an Iter pointing to
// some object does not prevent that object
// from being recycled.
//
template <class T> class Iter {
    T *ptr; // current pointer value
    T *end; // points to element one past end
    T *begin; // points to start of allocated array
    unsigned length; // length of sequence
public:

    Iter() {
        ptr = end = begin = NULL;
        length = 0;
    }

    Iter(T *p, T *first, T *last) {
        ptr = p;
        end = last;
        begin = first;
        length = last - first;
    }

```

```

// Return length of sequence to which this
// Iter points.
unsigned size() { return length; }

// Return value pointed to by ptr.
// Do not allow out-of-bounds access.
T &operator*() {
    if( (ptr >= end) || (ptr < begin) )
        throw OutOfRangeExc();
    return *ptr;
}

// Return address contained in ptr.
// Do not allow out-of-bounds access.
T *operator->() {
    if( (ptr >= end) || (ptr < begin) )
        throw OutOfRangeExc();
    return ptr;
}

// Prefix ++.
Iter operator++() {
    ptr++;
    return *this;
}

// Prefix --.
Iter operator--() {
    ptr--;
    return *this;
}

// Postfix ++.
Iter operator++(int notused) {
    T *tmp = ptr;
    ptr++;
    return Iter<T>(tmp, begin, end);
}

// Postfix --.
Iter operator--(int notused) {
    T *tmp = ptr;

    ptr--;
    return Iter<T>(tmp, begin, end);
}

// Return a reference to the object at the
// specified index. Do not allow out-of-bounds
// access.
T &operator[](int i) {
    if( (i < 0) || (i >= (end-begin)) )

```

```

        throw OutOfRangeExc();
    return ptr[i];
}

// Define the relational operators.
bool operator==(Iter op2) {
    return ptr == op2.ptr;
}

bool operator!=(Iter op2) {
    return ptr != op2.ptr;
}

bool operator<(Iter op2) {
    return ptr < op2.ptr;
}

bool operator<=(Iter op2) {
    return ptr <= op2.ptr;
}

bool operator>(Iter op2) {
    return ptr > op2.ptr;
}

bool operator>=(Iter op2) {
    return ptr >= op2.ptr;
}

// Subtract an integer from an Iter.
Iter operator-(int n) {
    ptr -= n;
    return *this;
}

// Add an integer to an Iter.
Iter operator+(int n) {
    ptr += n;
    return *this;
}

// Return number of elements between two Iters.
int operator-(Iter<T> &itr2) {
    return ptr - itr2.ptr;
}
};

// This class defines an element that is stored
// in the garbage collection information list.
//
template <class T> class GCInfo {
public:

```

```

unsigned refcount; // current reference count

T *memPtr; // pointer to allocated memory

/* isArray is true if memPtr points
   to an allocated array. It is false
   otherwise. */
bool isArray; // true if pointing to array

/* If memPtr is pointing to an allocated
   array, then arraySize contains its size */
unsigned arraySize; // size of array

// Here, mPtr points to the allocated memory.
// If this is an array, then size specifies
// the size of the array.
GCInfo(T *mPtr, unsigned size=0) {
    refcount = 1;
    memPtr = mPtr;
    if(size != 0)
        isArray = true;
    else
        isArray = false;

    arraySize = size;
}

};

// Overloading operator== allows GCInfos to be compared.
// This is needed by the STL list class.
template <class T> bool operator==(const GCInfo<T> &ob1,
                                   const GCInfo<T> &ob2) {
    return (ob1.memPtr == ob2.memPtr);
}

// GCPtr implements a pointer type that uses
// garbage collection to release unused memory.
// A GCPtr must only be used to point to memory
// that was dynamically allocated using new.
// When used to refer to an allocated array,
// specify the array size.
//
template <class T, int size=0> class GCPtr {

    // gclist maintains the garbage collection list.
    static list<GCInfo<T> > gclist;

    // addr points to the allocated memory to which
    // this GCPtr pointer currently points.
    T *addr;

```

```

/* isArray is true if this GCPtr points
   to an allocated array. It is false
   otherwise. */
bool isArray; // true if pointing to array

// If this GCPtr is pointing to an allocated
// array, then arraySize contains its size.
unsigned arraySize; // size of the array

// These support multithreading.
unsigned tid; // thread id
static HANDLE hThrd; // thread handle
static HANDLE hMutex; // handle of mutex
static int instCount; // counter of GCPtr objects

// Return an iterator to pointer info in gclist.
typename list<GCInfo<T> >::iterator findPtrInfo(T *ptr);

public:

// Define an iterator type for GCPtr<T>.
typedef Iter<T> GCiterator;

// Construct both initialized and uninitialized objects.
GCPtr(T *t=NULL) {

    // When first object is created, create the mutex
    // and register shutdown().
    if(hMutex==0) {
        hMutex = CreateMutex(NULL, 0, NULL);
        atexit(shutdown);
    }

    if(WaitForSingleObject(hMutex, 10000)==WAIT_TIMEOUT)
        throw TimeOutExc();

    list<GCInfo<T> >::iterator p;

    p = findPtrInfo(t);

    // If t is already in gclist, then
    // increment its reference count.
    // Otherwise, add it to the list.
    if(p != gclist.end())
        p->refcount++; // increment ref count
    else {
        // Create and store this entry.
        GCInfo<T> gcObj(t, size);
        gclist.push_front(gcObj);
    }

    addr = t;

```

```

arraySize = size;
if(size > 0) isArray = true;
else isArray = false;

// Increment instance counter for each new object.
instCount++;
// If the garbage collection thread is not
// currently running, start it running.
if(hThrd==0) {
    hThrd = (HANDLE) _beginthreadex(NULL, 0, gc,
        (void *) 0, 0, (unsigned *) &tid);

    // For some applications, it will be better
    // to lower the priority of the garbage collector
    // as shown here:
    //
    // SetThreadPriority(hThrd,
    //                     THREAD_PRIORITY_BELOW_NORMAL);
}

ReleaseMutex(hMutex);
}

// Copy constructor.
GCPtr(const GCPtr &ob) {
    if(WaitForSingleObject(hMutex, 10000)==WAIT_TIMEOUT)
        throw TimeOutExc();

    list<GCInfo<T> >::iterator p;

    p = findPtrInfo(ob.addr);
    p->refcount++; // increment ref count

    addr = ob.addr;
    arraySize = ob.arraySize;
    if(arraySize > 0) isArray = true;
    else isArray = false;

    instCount++; // increase instance count for copy

    ReleaseMutex(hMutex);
}

// Destructor for GCPtr.
~GCPtr();

// Collect garbage. Returns true if at least
// one object was freed.
static bool collect();

// Overload assignment of pointer to GCPtr.
T *operator=(T *t);

```

```

// Overload assignment of GCPtr to GCPtr.
GCPtr &operator=(GCPtr &rv);

// Return a reference to the object pointed
// to by this GCPtr.
T &operator*() {
    return *addr;
}

// Return the address being pointed to.
T *operator->() { return addr; }

// Return a reference to the object at the
// index specified by i.
T &operator[](int i) {
    return addr[i];
}

// Conversion function to T *.
operator T *() { return addr; }

// Return an Iter to the start of the allocated memory.
Iter<T> begin() {
    int size;

    if(isArray) size = arraySize;
    else size = 1;

    return Iter<T>(addr, addr, addr + size);
}

// Return an Iter to one past the end of an allocated array.
Iter<T> end() {
    int size;

    if(isArray) size = arraySize;
    else size = 1;

    return Iter<T>(addr + size, addr, addr + size);
}

// Return the size of gclist for this type
// of GCPtr.
static int gclistSize() {
    if(WaitForSingleObject(hMutex, 10000)==WAIT_TIMEOUT)
        throw TimeOutExc();

    unsigned sz = gclist.size();

    ReleaseMutex(hMutex);
    return sz;
}

```

```

}

// A utility function that displays gclist.
static void showlist();

// The following functions support multithreading.
//
// Returns true if the collector is still in use.
static bool isRunning() { return instCount > 0; }

// Clear gclist when program exits.
static void shutdown();

// Entry point for garbage collector thread.
static unsigned __stdcall gc(void * param);
};

// Create storage for the static variables.
template <class T, int size>
list<GCInfo<T> > GCPtr<T, size>::gclist;

template <class T, int size>
int GCPtr<T, size>::instCount = 0;

template <class T, int size>
HANDLE GCPtr<T, size>::hMutex = 0;

template <class T, int size>
HANDLE GCPtr<T, size>::hThrd = 0;

// Destructor for GCPtr.
template <class T, int size>
GCPtr<T, size>::~GCPtr() {
    if(WaitForSingleObject(hMutex, 10000) == WAIT_TIMEOUT)
        throw TimeOutExc();

    list<GCInfo<T> >::iterator p;
    p = findPtrInfo(addr);
    if(p->refcount) p->refcount--; // decrement ref count

    // Decrement instance counter for each object
    // that is destroyed.
    instCount--;

    ReleaseMutex(hMutex);
}

// Collect garbage. Returns true if at least
// one object was freed.
template <class T, int size>
bool GCPtr<T, size>::collect() {
    if(WaitForSingleObject(hMutex, 10000) == WAIT_TIMEOUT)

```



```

        throw TimeOutExc();

    bool memfreed = false;

    list<GCInfo<T> >::iterator p;
    do {

        // Scan gclist looking for unreferenced pointers.
        for(p = gclist.begin(); p != gclist.end(); p++) {
            // If in-use, skip.
            if(p->refcount > 0) continue;

            memfreed = true;

            // Remove unused entry from gclist.
            gclist.remove(*p);

            // Free memory unless the GCPtr is null.
            if(p->memPtr) {
                if(p->isArray) {
                    delete[] p->memPtr; // delete array
                }
                else {
                    delete p->memPtr; // delete single element
                }
            }

            // Restart the search.
            break;
        }
    } while(p != gclist.end());

    ReleaseMutex(hMutex);

    return memfreed;
}

// Overload assignment of pointer to GCPtr.
template <class T, int size>
T * GCPtr<T, size>::operator=(T *t) {
    if(WaitForSingleObject(hMutex, 10000)==WAIT_TIMEOUT)
        throw TimeOutExc();

    list<GCInfo<T> >::iterator p;

    // First, decrement the reference count
    // for the memory currently being pointed to.
    p = findPtrInfo(addr);
    p->refcount--;

    // Next, if the new address is already
    // existent in the system, increment its

```

```

    // count. Otherwise, create a new entry
    // for gclist.
    p = findPtrInfo(t);
    if(p != gclist.end())
        p->refcount++;
    else {
        // Create and store this entry.
        GCInfo<T> gcObj(t, size);
        gclist.push_front(gcObj);
    }

    addr = t; // store the address.

    ReleaseMutex(hMutex);

    return t;
}

// Overload assignment of GCPtr to GCPtr.
template <class T, int size>
GCPtr<T, size> & GCPtr<T, size>::operator=(GCPtr &rv) {
    if(WaitForSingleObject(hMutex, 10000)==WAIT_TIMEOUT)
        throw TimeOutExc();
    list<GCInfo<T> >::iterator p;

    // First, decrement the reference count
    // for the memory currently being pointed to.
    p = findPtrInfo(addr);
    p->refcount--;

    // Next, increment the reference count
    // of the new object.
    p = findPtrInfo(rv.addr);
    p->refcount++; // increment ref count

    addr = rv.addr; // store the address.

    ReleaseMutex(hMutex);

    return rv;
}

// A utility function that displays gclist.
template <class T, int size>
void GCPtr<T, size>::showlist() {
    if(WaitForSingleObject(hMutex, 10000)==WAIT_TIMEOUT)
        throw TimeOutExc();

    list<GCInfo<T> >::iterator p;

    cout << "gclist<" << typeid(T).name() << ", "
         << size << ">:\n";
}

```

```

    cout << "memPtr refcount value\n";

    if(gclist.begin() == gclist.end()) {
        cout << " -- Empty --\n\n";
        return;
    }

    for(p = gclist.begin(); p != gclist.end(); p++) {
        cout << "[" << (void *)p->memPtr << "]"
            << " " << p->refcount << " ";
        if(p->memPtr) cout << " " << *p->memPtr;
        else cout << " ---";
        cout << endl;
    }
    cout << endl;

    ReleaseMutex(hMutex);
}

// Find a pointer in gclist.
template <class T, int size>
typename list<GCInfo<T> >::iterator
    GCPtr<T, size>::findPtrInfo(T *ptr) {

    list<GCInfo<T> >::iterator p;

    // Find ptr in gclist.
    for(p = gclist.begin(); p != gclist.end(); p++)
        if(p->memPtr == ptr)
            return p;

    return p;
}

// Entry point for garbage collector thread.
template <class T, int size>
unsigned __stdcall GCPtr<T, size>::gc(void * param) {
    #ifdef DISPLAY
        cout << "Garbage collection started.\n";
    #endif

    while(isRunning()) {
        collect();
    }

    collect(); // collect garbage on way out

    // Release and reset the thread handle so
    // that the garbage collection thread can
    // be restarted if necessary.
    CloseHandle(hThrd);
    hThrd = 0;
}

```

```

#ifdef DISPLAY
    cout << "Garbage collection terminated for "
          << typeid(T).name() << "\n";
#endif

    return 0;
}

// Clear gclist when program exits.
template <class T, int size>
void GCPtr<T, size>::shutdown() {

    if(gclistSize() == 0) return; // list is empty

    list<GCInfo<T> >::iterator p;

#ifdef DISPLAY
        cout << "Before collecting for shutdown() for "
              << typeid(T).name() << "\n";
#endif

    for(p = gclist.begin(); p != gclist.end(); p++) {
        // Set all remaining reference counts to zero.
        p->refcount = 0;
    }

    collect();

#ifdef DISPLAY
        cout << "After collecting for shutdown() for "
              << typeid(T).name() << "\n";
#endif
}

```

3.6.10 多线程垃圾回收器的使用

为了使用多线程的垃圾回收器，需要在您的程序中包含 `gcthrd.h`。然后，以第 2 章描述的方式使用 `GCPtr`。当编译这个程序时，必须要记住链接多线程库，如同本章前面对 `_beginthreadex()` 和 `_endthreadex()` 的描述中所讲述的那样。

为了观察多线程垃圾回收器的效果，尝试原先在第 2 章中给出的这个版本的加载测试程序：

```

// Demonstrate the multithreaded garbage collector.
#include <iostream>
#include <new>
#include "gcthrd.h"

using namespace std;

// A simple class for load testing GCPtr.
class LoadTest {

```

```

    int a, b;
public:
    double n[100000]; // just to take-up memory
    double val;

    LoadTest() { a = b = 0; }

    LoadTest(int x, int y) {
        a = x;
        b = y;
        val = 0.0;
    }

    friend ostream &operator<<(ostream &strm, LoadTest &obj);
};

// Create an inserter for LoadTest.
ostream &operator<<(ostream &strm, LoadTest &obj) {
    strm << "(" << obj.a << " " << obj.b << ")";
    return strm;
}

int main() {
    GCPtr<LoadTest> mp;
    int i;

    for(i = 1; i < 2000; i++) {
        try {
            mp = new LoadTest(i, i);
            if(!(i%100))
                cout << "gclist contains " << mp.gclistSize()
                     << " entries.\n";
        } catch(bad_alloc xa) {
            // For most users, this exception won't
            // ever occur.
            cout << "Last object: " << *mp << endl;
            cout << "Length of gclist: "
                 << mp.gclistSize() << endl;
        }
    }

    return 0;
}

```

在此有一个运行的样本。(当然, 您的输出可能会不同)。在 `gcthrd.h` 中定义了 `DISPLAY` 宏并开启显示选项时, 输出如下:

```

Garbage collection started.
gclist contains 42 entries.
gclist contains 35 entries.
gclist contains 29 entries.
gclist contains 22 entries.

```

```

gclist contains 18 entries.
gclist contains 11 entries.
gclist contains 4 entries.
gclist contains 51 entries.
gclist contains 47 entries.
gclist contains 40 entries.
gclist contains 33 entries.
gclist contains 26 entries.
gclist contains 19 entries.
gclist contains 15 entries.
gclist contains 10 entries.
gclist contains 3 entries.
gclist contains 53 entries.
gclist contains 46 entries.
gclist contains 42 entries.
Before collecting for shutdown() for class LoadTest
After collecting for shutdown() for class LoadTest

```

正如您所看到的那样，由于 `collect()` 在后台运行，虽然有成千个对象被分配并丢弃，但是 `gclist` 也不会非常大。

3.7 试着完成下面的任务

创建一个成功的多线程程序具有相当的挑战性。原因之一是多线程要求您用并行的而不是线性的观点来考虑程序。另外，在运行时，线程交互的方式通常难以预测。因此，您可能会对多线程程序的操作感到惊奇(甚至迷惑)。掌握多线程的最好方法是使用它。为此，您或许会尝试这些想法。

试着在线程控制面板内添加另外一个列表框，以使得用户除了调整线程的优先级值之外，还可以调整线程的优先级类别。试着在控制面板内加入不同的同步对象，这些对象可以在用户的控制下开启或者关闭。这将会让您体验不同的同步选择。

对于多线程的垃圾回收器，试着减少垃圾回收的次数，如 `gclist` 达到某个大小时或者在自由内存低于某个预定值时才进行回收。作为另一种选择，可以使用可等待计时器有规律地激活垃圾回收。最后，您可以试验不同的垃圾回收器的优先级类别和设置来找到最适合您使用的优先级。

第 4 章 C++ 的扩展

对于典型的 C++ 程序员，程序设计不只是工作，还是一种生活方式。程序员不只是需要语言的原始功能，还需要它的细微与精妙。就像一个品酒师品味葡萄酒一样，我们也享受语言出色的特征。当然，我们对 C++ 的兴趣不会只局限于用它来编写程序。相反，通常我们发现自己被语言本身以及围绕语言的设计和开发的问题所吸引。

大多数 C++ 程序员都对计算机语言有兴趣，很少有 C++ 程序员没有梦想过对语言加入新的功能。(您想过多少次“如果 C++ 能够……该多好啊”?) 问题是大多数程序员都不能够访问完善的 C++ 编译器来加入他们的实验性结构。幸运的是，在此有一个简单的方法来试验您自己对 C++ 的扩展：通过创建一个译码器(translator)，将您的想法转换为实际的 C++ 代码。这种译码器是本章的主题。

4.1 为什么使用译码器

由于在此有一个明显的选择：选择标准 C++ 预处理器，您可能会问，为什么需要使用译码器来体验新的语言特性。大家都知道，预处理器支持宏替换，其中可以用一个文本序列来替换另一个文本序列。许多年来，预处理器宏一直被用来向 C++ 中加入新的特性。例如，宏经常实现的一个结构是 repeat/until 循环。repeat/until 循环类似于 C++ 的 do/while 循环，只是 repeat/until 循环只有在条件变为 true 时才会终止。(也就是说，repeat/until 在条件为 false 时运行)。这不同于 do/while 循环，它在条件变为 false 时终止。由于 repeat/until 循环和 do/while 循环之间的相似性，可以用宏来实现 repeat/until 循环，如下所示：

```
#define repeat do
#define until(exp) while(!{exp})
// ...
int i=0;
repeat {
    cout << "i: " << i << endl;
    i++;
} until(i==10);
```

这个宏使得 repeat 取代 do，until 取代 while。另外，在 while 中的条件表达式被反转。因此，预处理器会将前面代码段中的 repeat/until 循环转换为这个 do/while 循环：

```
do {
    cout << "i: " << i << endl;
    i++;
} while(!(i==10));
```

正如这个示例显示的，使用宏可以轻松合理地实现 repeat/until 结构。

既然使用宏实现新的特性已经有好多年的历史，那为什么要创建译码器呢？也就是说，为什么不像刚才对 repeat/until 那样使用宏呢？答案是，并不能使用宏替换来加入所有类型的新特性。另外，即使可以使用宏替换，也并不是所有的替换都像 repeat/until 示例那样优雅并且在概念上很完美。尽管程序员已经通过宏实现了很多令人瞩目的特性，但是这些任务在很多情况下是通过使用复杂的、难以理解的#define 指示来完成的。这种“宏陷阱”经常会导致结构不好的代码，很难验证其正确性，并且缺乏弹性。坦白地说，由于上面的原因，许多 C++ 程序员(包括作者)避免使用复杂的宏。

幸运的是，在此有一个对复杂的宏的替代方案，可以用它试验多种新的 C++ 特性：可以创建一个译码器，在程序的控制之下将试验结构转换为等价的 C++ 代码。使用这种“提前预处理器”可以向 C++ 加入使用预处理器宏很难实现甚至不可能实现的特性。这个译码器本身就是一个有趣的项目，并且可以很容易地修改这个框架来适应其他需要。

尽管本章开发的译码器能够执行复杂的文本替换，但它仍然有其局限性，理解这一点很重要。由于这个译码器实现了单向算法，只读取一次源文件，因此它只能用于单向替换就可以处理的结构。(如果您愿意的话，可以修改这个特性)。除了执行复杂的文本替换之外，译码器完全忽略程序的内容。也就是说，它对变量的类型、运算符的意义，甚至先前读取的内容一无所知。因此，这个译码器并不是 C++ 语言的分析器。它只是对指定文本替换的一个引擎；在本质上，它是一个特殊的预处理器。尽管有这些限制，仍然可以使用它试验各种各样的想法。

提示：

如果您对 C++ 的分析器感兴趣，请参考第 9 章。

4.2 实验性的关键字

在开发译码器之前，有必要定义它所执行的转换。本章开发的译码器处理如下的对 C++ 实验性的扩展：

- 一个 foreach 循环
- 一条 cases 语句
- 一个 typeof 运算符
- 一个 repeat/until 循环

除了 repeat/until 循环之外(包含它只是为了演示)，其他的关键字都使用了宏替换很难(甚至不可能)转换的语法。对每个实验性关键字的描述如下。

4.2.1 foreach 循环

当前开发的语言已经包含了 foreach 循环。例如，foreach 是 C# 的一部分，Java 中也加入了“for each”风格的循环。另外，“for each”的思想已经被整合为最新 C++ 的一部分，因为 STL(标准模板库)定义了 for_each() 算法，对容器中的每个元素都应用了一个函数。然而，C++ 目前还没有定义多用途的 foreach 循环。

foreach 用来处理程序设计中经常遇到的情况：需要按照严格的顺序从头到尾遍历数组的元素(或者其他类型的对象集合)。例如，考虑这个计算数组元素平均值的代码段：


```
int n[] = { 1, 7, 3, 11, 5 };
double avg=0.0;

for(int i=0; i < 5; i++)
    avg += n[i];

cout << "Average: " << avg/5 << endl;
```

为了计算 `n` 的平均值，按顺序从头到尾读取了数组的每个元素。当然，这只是普遍概念的一个实例。可以使用相似类型的循环来查找数组的最大值或者最小值、对数组的值求和、计算最小公约数，可能还有上百个其他的用途。另外，当需要访问数组的内容时，都可以使用相同类型的循环结构。`foreach` 循环就是为了简化并改进这类循环而创建的。

`foreach` 的价值在于它取消了手工索引数组的需要。相反，`foreach` 自动遍历整个数组，按顺序在一个时刻获取一个元素。例如，下面是用 `foreach` 修改过的前面的代码段：

```
int n[] = { 1, 7, 3, 11, 5 };
double avg=0.0;

foreach(int x in n)
    avg += x;

cout << "Average: " << avg/5 << endl;
```

每循环一次，自动赋予 `x` 的值等于 `n` 中下一个元素的值。因此，第一次迭代 `x` 的值为 1，第二次迭代 `x` 的值为 7，以此类推。从而不仅对语法进行了改进，同时也防止了计数错误。

`foreach` 循环的语法如下：

```
foreach(type varname in arrayname)
```

在此，`type` 为循环变量的类型，其名称由 `varname` 指定，被访问的数组由 `arrayname` 指定。另外，`varname` 局限于循环之内，在循环外不会被知道(这个语法由 C# 而来)。记住，循环的每次迭代过程中，`varname` 都会包含(按顺序)指定数组的下一个元素的值。

译码器将 `foreach` 循环转换为其等价的 C++ `for` 循环。

4.2.2 cases 语句

`cases` 语句允许您指定一个值的范围来与 `switch` 表达式匹配。通常，当您想让两个或者多个 `case` 语句使用相同的代码序列时，必须使用“堆砌的”`case` 语句，下面的示例显示了这一点：

```
switch(i) {
    case 1:
    case 2:
    case 3:
    case 4:
        // do something for cases 1 to 4
        break;
    case 5:
        // do something else
        break;
    // ...
}
```

堆砌的 `case` 语句(如这个示例中的 1 到 4 条)在程序设计中很常见。这是乏味的代码。为什么不在一个 `case` 内指定一个值的范围来简化呢? 译码器正好可以让您做到这一点!

译码器实现了一个 `cases` 语句, 使您可以定义一个值的范围, 在一个代码序列中处理范围内的所有值。例如, 先前的代码可以使用 `cases` 语句来修改:

```
switch(i) {
  cases 1 to 4:
    // do something for cases 1 to 4
    break;
  case 5:
    // do something else
    break;
  // ...
}
```

这样, 当 `i` 包括在 1 到 4 之间时, 这个值将由 `cases` 语句来匹配。

`cases` 语句的语法如下所示:

```
cases start to end;
```

在此, `start` 是用来匹配的起始值, `end` 是用来匹配的结束值。

译码器将 `cases` 语句转换为一系列堆砌的 `case` 语句。

4.2.3 `typeid` 运算符

运行时类型 ID 已经成为大多数现代程序设计重要的一部分。尽管 C++ 内建的对它的支持很不错, 但是程序员仍然不停地试图寻求更好的性能。`typeid` 实验性运算符就是一个例子。它只是提供了 C++ 已经支持的操作的另一个可选择的语法: 两个类型的比较。因此, `typeid` 没有增加新的功能, 但是它提供了这个过程的不同观点。

通常, 当需要类型比较时, 会使用 `typeid` 运算符。例如, 下面的语句用来判断 `ptr1` 所指的對象是否与 `ptr2` 指的對象具有相同的类型:

```
if(typeid(*ptr1) == typeid(*ptr2))
  cout << "ptr1 points to same type as ptr2\n";
```

在这条语句中, `typeid` 运算符获取了 `ptr1` 和 `ptr2` 所指对象的类型。如果这两个类型相同, 则 `if` 语句成功。当使用多态性类时, 可能会用到这样的语句。在此情况下, 基类指针所指对象的类型不能在编译时得知, 从而需要运行时检查。

尽管 `typeid` 或前面的语句没有错误, 但是下面的方法提供了一种更为有趣的可选语法:

```
if(typeof *ptr1 same as *ptr2)
  cout << "ptr1 points to same type as ptr2\n";
```

在这条语句中, `typeof` 运算符用来比较两种类型。如果这两种类型相同, 则返回 `true`; 否则返回 `false`。尽管它执行的操作与第一个版本相同, 但它改变了操作的表达方式, 从而使得您以不同的观点来看待它。它还说明了译码器让您可以体验的想法的范围。

`typeof` 的语法如下:

```
typeof op1 same as op2
```

在此, *op1* 和 *op2* 指定了类型标识符(如 `int` 或者 `MyClass`)或者对象。因此, `typeof` 可以用来比较两个对象的类型, 一个对象的类型和一个已知类型, 或者两个类型。

译码器将 `typeof` 转换为其对应的 `typeid` 表达式。

4.2.4 repeat/until 循环

如前所述, 很容易使用预处理器宏来实现 `repeat/until` 循环。用译码器来实现 `repeat/until` 只是为了说明问题, 并且由于它可以作为其他类型循环的模型, 您或许想要试验这些新的循环。

4.3 试验 C++ 新特性的译码器

译码器的全部代码如下所示。为了前后一致, 将这个文件命名为 `trans.cpp`。

```
// A translator for experimental C++ extensions.
#include <iostream>
#include <fstream>
#include <cctype>
#include <cstring>
#include <string>

using namespace std;

// Prototypes for the functions that handle
// the extended keywords.
void foreach();
void cases();
void repeat();
void until();
void typeof();

// Prototypes for tokenizing the input file.
bool gettoken(string &tok);
void skipspaces();

// Indentation padding string.
string indent = "";

// The input and output file streams.
ifstream fin;
ofstream fout;

// Exception class for syntax errors.
class SyntaxExc {
    string what;
public:
    SyntaxExc(char *e) { what = string(e); }
    string geterror() { return what; }
};
```

```

int main(int argc, char *argv[]) {
    string token;

    if(argc != 3) {
        cout << "Usage: ep <input file> <output file>\n";
        return 1;
    }

    fin.open(argv[1]);

    if(!fin) {
        cout << "Cannot open " << argv[1] << endl;
        return 1;
    }

    fout.open(argv[2]);

    if(!fout) {
        cout << "Cannot open " << argv[2] << endl;
        return 1;
    }

    // Write header.
    fout << "// Translated from an .exp source file.\n";

    try {
        // Main translation loop.
        while(gettoken(token)) {

            // Skip over // comments.
            if(token == "//") {
                do {
                    fout << token;
                    gettoken(token);
                } while(token.find('\n') == string::npos);
                fout << token;
            }

            // Skip over /* comments.
            else if(token == "/*") {
                do {
                    fout << token;
                    gettoken(token);
                } while(token != "*/");
                fout << token;
            }

            // Skip over quoted string.
            else if(token == "\"") {
                do {
                    fout << token;

```

```

        gettoken(token);
    } while(token != "\\");
    fout << token;
}

else if(token == "foreach") foreach();

else if(token == "cases") cases();

else if(token == "repeat") repeat();

else if(token == "until") until();

else if(token == "typeof") typeof();

    else fout << token;
}
} catch(SyntaxExc exc) {
    cout << exc.geterror() << endl;
    return 1;
}

return 0;
}

// Get the next token from the input stream.
bool gettoken(string &tok) {
    char ch;
    char ch2;
    static bool trackIndent = true;

    tok = "";

    ch = fin.get();

    // Check for EOF and return false if EOF
    // is found.
    if(!fin) return false;

    // Read whitespace.
    if(isspace(ch)) {
        while(isspace(ch)) {
            tok += ch;

            // Reset indent counter with each new line.
            if(ch == '\n') {
                indent = "";
                trackIndent = true;
            }
            else if(trackIndent) indent += ch;

            ch = fin.get();

```

```

    }
    fin.putback(ch);
    return true;
}

// Stop tracking indentation after encountering
// first non-whitespace character on a line.
trackIndent = false;

// Read an identifier or keyword.
if(isalpha(ch) || ch=='_') {
    while(isalpha(ch) || isdigit(ch) || ch=='_') {
        tok += ch;
        ch = fin.get();
    }
    fin.putback(ch);
    return true;
}

// Read a number.
if(isdigit(ch)) {
    while(isdigit(ch) || ch=='.' ||
          tolower(ch) == 'e' ||
          ch == '-' || ch == '+') {
        tok += ch;
        ch = fin.get();
    }
    fin.putback(ch);
    return true;
}

// Check for \"
if(ch == '\\') {
    ch2 = fin.get();
    if(ch2 == '\"') {
        tok += ch;
        tok += ch2;
        ch = fin.get();
    } else
        fin.putback(ch2);
}

// Check for '''
if(ch == '\\') {
    ch2 = fin.get();
    if(ch2 == '\"') {
        tok += ch;
        tok += ch2;
        return true;
    } else
        fin.putback(ch2);
}

```

```

// Check for begin comment symbols.
if(ch == '/') {
    tok += ch;
    ch = fin.get();
    if(ch == '/' || ch == '*') {
        tok += ch;
    }
    else fin.putback(ch);
    return true;
}

// Check for end comment symbols.
if(ch == '*') {
    tok += ch;
    ch = fin.get();
    if(ch == '/') {
        tok += ch;
    }
    else fin.putback(ch);
    return true;
}

tok += ch;
return true;
}

// Translate a foreach loop.
void foreach() {
    string token;
    string varname;
    string arrayname;

    char forvarname[5] = "_i";
    static char counter[2] = "a";

    // Create loop control variable for generated
    // for loop.
    strcat(forvarname, counter);
    counter[0]++;

    // Only 26 foreach loops in a file because
    // generated loop control variables limited to
    // _ia to _iz. This can be changed if desired.
    if(counter[0] > 'z')
        throw SyntaxExc("Too many foreach loops.");

    fout << "int " << forvarname
        << " = 0;\n";

    // Write beginning of generated for loop.
    fout << indent << "for(";

```

```

    skipspace();

    // Read the (
    gettoken(token);
    if(token[0] != '(')
        throw SyntaxExc("( expected in foreach.");

    skipspace();

    // Get the type of the foreach variable.
    gettoken(token);
    fout << token << " ";

    skipspace();

    // Read and save the foreach variable's name.
    gettoken(token);
    varname = token;

    skipspace();

    // Read the "in"
    gettoken(token);
    if(token != "in")
        throw SyntaxExc("in expected in foreach.");

    skipspace();

    // Read the array name.
    gettoken(token);
    arrayname = token;

    fout << varname << " = " << arrayname << "[0];\n";

    // Construct target value.
    fout << indent + " " << forvarname << " < "
        << "((sizeof " << token << ")/"
        << "(sizeof " << token << "[0]));\n";

    fout << indent + " " << forvarname << "++,"
        << varname << " = " << arrayname << "["
        << forvarname << "])";

    skipspace();

    // Read the )
    gettoken(token);
    if(token[0] != ')')
        throw SyntaxExc(") expected in foreach.");
}

```



```

// Translate a cases statement.
void cases() {
    string token;
    int start, end;

    skipspaces();

    // Get starting value.
    gettoken(token);
    if(isdigit(token[0])) {
        // is an int constant
        start = atoi(token.c_str());
    }
    else if(token[0] == '\\') {
        // is char constant
        gettoken(token);

        start = (int) token[0];

        // discard closing '
        gettoken(token);
        if(token[0] != '\\')
            throw SyntaxExc("' expected in cases.");
    }
    else
        throw SyntaxExc("Constant expected in cases.");

    skipspaces();

    // Read and discard the "to".
    gettoken(token);
    if(token != "to")
        throw SyntaxExc("to expected in cases.");

    skipspaces();

    // Get ending value.
    gettoken(token);

    if(isdigit(token[0])) {
        // is an int constant
        end = atoi(token.c_str());
    }
    else if(token[0] == '\\') {
        // is char constant
        gettoken(token);

        end = (int) token[0];

        // discard closing '
        gettoken(token);
        if(token[0] != '\\')

```

```

        throw SyntaxExc("' expected in cases.");
    }
    else
        throw SyntaxExc("Constant expected in cases.");

    skipspace();

    // Read and discard the :
    gettoken(token);

    if(token != ":")
        throw SyntaxExc(": expected in cases.");

    // Generate stacked case statements.
    fout << "case " << start << ":\n";
    for(int i = start+1 ; i <= end; i++) {
        fout << indent << "case " << i << ":";
        if(i != end) fout << endl;
    }
}

// Translate a repeat loop.
void repeat() {
    fout << "do";
}

// Translate an until.
void until() {
    string token;
    int parencount = 1;

    fout << "while";

    skipspace();

    // Read and store the (
    gettoken(token);
    if(token != "(")
        throw SyntaxExc("(" expected in typeof.");
    fout << "(";

    // Begin while by reversing and
    // parenthesizing the condition.
    fout << "!(";

    // Now, read the expression.
    do {
        if(!gettoken(token))
            throw SyntaxExc("Unexpected EOF encountered.");

        if(token == "(") parencount++;
        if(token == ")") parencount--;
    }

```

```

    fout << token;
} while(parencount > 0);
fout << ")";
}

// Translate a typeof expression.
void typeof() {
    string token;
    string temp;

    fout << "typeid(";

    skipspaces();

    gettoken(token);

    do {
        temp = token;

        if(!gettoken(token))
            throw SyntaxExc("Unexpected EOF encountered.");

        if(token != "same") fout << temp;
    } while(token != "same");

    skipspaces();

    gettoken(token);

    if(token != "as") throw SyntaxExc("as expected.");

    fout << ") == typeid(";

    skipspaces();

    do {
        if(!gettoken(token))
            throw SyntaxExc("Unexpected EOF encountered.");

        fout << token;
    } while(token != ")");
    fout << ")";
}

void skipspaces() {
    char ch;

    do {
        ch = fin.get();
    } while(isspace(ch));
    fin.putback(ch);
}

```

4.4 使用译码器

为了使用译码器，首先要创建一个文件，其中包含使用了实验性关键字的程序。为了直观起见，将使用了实验性结构的文件的扩展名定为.exp。需要理解的是，exp 文件主要包含标准 C++ 代码。它只是还包含了一个或者多个实验性的特征，这些特征会被译码器转换为 C++ 代码。例如，下面是用来说明 foreach 循环的 exp 文件。注意大部分程序还是由普通的 C++ 代码组成。

```
// Demonstrate the foreach loop.
#include <iostream>
using namespace std;

int main() {
    int nums[] = { 1, 6, 19, 4, -10, 88 };
    int min;

    // Find the minimum value.
    min = nums[0];
    foreach(int x in nums)
        if(min > x) min = x;

    cout << "Minimum is " << min << endl;

    return 0;
}
```

为了将 exp 文件转换为标准的 cpp 文件，要通过译码器运行它。例如，假定这个文件的名称为 foreach.exp，下面的命令行将把它转化为可以被 C++ 编译器编译的纯粹的 C++ 代码：

```
trans foreach.exp foreach.cpp
```

在 trans 运行之后，foreach.cpp 将包含如下的 C++ 程序：

```
// Translated from an .exp source file.
// Demonstrate the foreach loop.
#include <iostream>
using namespace std;

int main() {
    int nums[] = { 1, 6, 19, 4, -10, 88 };
    int min;

    // Find the minimum value.
    min = nums[0];
    int _ia = 0;
    for(int x = nums[0];
        _ia < ((sizeof nums)/(sizeof nums[0]));
        _ia++, x = nums[_ia])
        if(min > x) min = x;

    cout << "Minimum is " << min << endl;
```

```
    return 0;
}
```

本章的剩余部分将讨论译码器的运行方式。

4.5 译码器的运行方式

译码器以直接的方式操作。它只是读取输入文件并将之编写为输出文件。在此过程中，当发现实验性关键字时，就会将其转换为等价的 C++ 代码。这种内在的简单性就是译码器适合于试验的原因。不需要花很大的力气来实现一个扩展，然后再测试它。下面的几节将详细讨论译码器的各个部分。

4.5.1 全局声明

译码器在开始定义了如下的全局变量和类：

```
// Indentation padding string.
string indent = "";
// The input and output file streams.
ifstream fin;
ofstream fout;

// Exception class for syntax errors.
class SyntaxExc {
    string what;
public:
    SyntaxExc(char *e) { what = string(e); }
    string geterror() { return what; }
};
```

当前用来缩进的空白序列存储在 `indent` 中。当为试验结构替换多行代码时，这个字符串用来加入数量合适的缩进。

输入文件流存储在 `fin` 中；输出文件流保存在 `fout` 中。当程序开始时，命令行指定的文件名与 `fin` 和 `fout` 链接。

如果在试验结构的转换中出现了语法错误，就会抛出 `SyntaxExc` 对象来报告。如同它的字面意思那样，`SyntaxExc` 值包含了描述错误的字符串，但如果需要的话，可以加入新的功能。

4.5.2 `main()` 函数

`main()` 函数执行两个任务。首先，它打开命令行指定的输入和输出文件。C++ 程序员应该很熟悉这些代码。其次，它运行译码器的主循环，这个循环处理试验关键字的转换。主循环如下所示：

```
try {
    // Main translation loop.
    while(gettoken(token)) {
```

```

// Skip over // comments.
if(token == "//") {
    do {
        fout << token;
        gettoken(token);
    } while(token.find('\n') == string::npos);
    fout << token;
}

// Skip over /* comments.
else if(token == "/*") {
    do {
        fout << token;
        gettoken(token);
    } while(token != "*/");
    fout << token;
}

// Skip over quoted string.
else if(token == "\"") {
    do {
        fout << token;
        gettoken(token);
    } while(token != "\"");
    fout << token;
}

else if(token == "foreach") foreach();

else if(token == "cases") cases();

else if(token == "repeat") repeat();

else if(token == "until") until();

else if(token == "typeof") typeof();

else fout << token;
}
} catch(SyntaxExc exc) {
    cout << exc.geterror() << endl;
    return 1;
}
}

```

每次循环都会从输入文件中读入一个令牌。如果这个令牌不需要转换,就将其写入到输出文件。然而,如果这个令牌包含了一个实验性关键字,就会调用合适的函数将这个关键字转换为对应的C++代码。注意,这个循环忽略了注释以及引号中的字符串,并将其复制到输出文件中,而没有检查它们的内容中是否包含实验性关键字。为了阻止对注释中的关键字或者引号中字符串中包含的关键字的转换,这样做是有必要的。

如果在试验性特征的转换中发生了语法错误,执行转换的代码会抛出 `SyntaxExc` 异常,这

个异常会被 `main()` 中的 `catch` 捕获，它只是简单地显示这个错误。您可以增强这个错误报告，如果您需要的话，可以在其中包含行数或者其他信息。

4.5.3 `gettoken()` 和 `skipspaces()` 函数

为了将实验性的关键字转换为等价的 C++ 代码，译码器必须知道何时发现了一个关键字。为此，输入文件必须由一个令牌接一个令牌地处理。在此，术语标记是一个很松散的概念，它只是意味着一小段文本。译码器并不需要处理全部的令牌，它只需要认出标识符(包括关键字)、数字以及空白。大多数其他的字符可以作为单个字符来处理。

`gettoken()` 函数如下所示：

```
// Get the next token from the input stream.
bool gettoken(string &tok) {
    char ch;
    char ch2;
    static bool trackIndent = true;

    tok = "";

    ch = fin.get();

    // Check for EOF and return false if EOF
    // is found.
    if(!fin) return false;

    // Read whitespace.
    if(isspace(ch)) {
        while(isspace(ch)) {
            tok += ch;

            // Reset indent counter with each new line.
            if(ch == '\n') {
                indent = "";
                trackIndent = true;
            }
            else if(trackIndent) indent += ch;

            ch = fin.get();
        }
        fin.putback(ch);
        return true;
    }

    // Stop tracking indentation after encountering
    // first non-whitespace character on a line.
    trackIndent = false;
    // Read an identifier or keyword.
    if(isalpha(ch) || ch=='_') {
        while(isalpha(ch) || isdigit(ch) || ch=='_') {
            tok += ch;
```

```

        ch = fin.get();
    }
    fin.putback(ch);
    return true;
}

// Read a number.
if(isdigit(ch)) {
    while(isdigit(ch) || ch=='.' ||
          tolower(ch) == 'e' ||
          ch == '-' || ch == '+') {
        tok += ch;
        ch = fin.get();
    }
    fin.putback(ch);
    return true;
}

// Check for \"
if(ch == '\\') {
    ch2 = fin.get();
    if(ch2 == '"') {
        tok += ch;
        tok += ch2;
        ch = fin.get();
    } else
        fin.putback(ch2);
}

// Check for '''
if(ch == '\\') {
    ch2 = fin.get();
    if(ch2 == '\'') {
        tok += ch;
        tok += ch2;
        return true;
    } else
        fin.putback(ch2);
}

// Check for begin comment symbols.
if(ch == '/') {
    tok += ch;
    ch = fin.get();
    if(ch == '/' || ch == '*') {
        tok += ch;
    }
    else fin.putback(ch);
    return true;
}

// Check for end comment symbols.

```



```

    if(ch != '*') {
        tok += ch;
        * ch = fin.get();
        if(ch == '/') {
            tok += ch;
        }
        else fin.putback(ch);
        return true;
    }

    tok += ch;

    return true;
}

```

`gettoken()`函数有一个参数，一个名为 `tok` 的字符串，传递一个 `string` 对象的引用。当函数返回时，这个对象将包含令牌。如果某个令牌被读取，则函数返回 `true`；如果碰到文件结尾，则返回 `false`。

`gettoken()`函数开始将 `tok` 设置为 `null` 字符串。然后从 `fin` 中读取下一个字符，并将其存储在 `ch` 中。如果读取时碰到了文件结尾，则返回 `false`。否则，会测试 `ch` 的各种可能性。

首先，如果 `ch` 为空白，就会进入一个循环来读取空白字符，直到读取了第一个非空白字符为止。空白字符追加在 `tok` 后面。通过调用 `fin.putback()`将非空白字符返回到输入流中。在循环的结尾，`tok` 包含了所有被读取的空白字符，这就是返回到调用例程的标记。

在空白循环中还有其他一些事情需要注意。如果读取新的一行，`indent` 则被设置为 `null` 字符串，`trackIndent` 变量被设置为 `true`。当 `trackIndent` 为 `true` 时，空白将存储在 `indent` 中。在空白循环之后，`trackIndent` 被设置为 `false`。因此，在 `indent` 中只存储一行开头的空白。

如果 `ch` 不是空白字符，就会测试其他的可能性。如果下一个标记是标识符或者关键字，它以字母或者下划线开始，并且被 `gettoken()`中的下一个循环读取。否则，如果 `ch` 是一个数字，就会读取一个数值。

如果 `ch` 不是空白、字母、下划线或者数字，那么就会检查 5 种特殊的状态。第一个是“`\`”序列。如前所述，译码器不会对引号内的文本执行转换。当译码器发现开引号时，只是简单地将其与闭引号之间的文本复制。然而，有必要阻止嵌套的引号被误认为是引号内字符串的开始或者结尾。因此，反斜杠序列“`\`”必须被作为一对字符来处理。当某个字符常量指定一个引号时，情况相同。注释符号也必须作为一对字符来处理，因此 `gettoken()`检查 `//`、`/*`，以及 `*/`。

当执行转换时，有些情况下必须抛弃实验性语句中的空白，因为它们与生成的 C++ 代码没有关系。在此显示的 `skipspaces()`函数可以完成这个任务。它只是读取并舍弃空白。

```

void skipspaces() {
    char ch;

    do {
        ch = fin.get();
    } while(isspace(ch));
    fin.putback(ch);
}

```

4.5.4 转换 foreach 循环

最具有挑战性的是对 foreach 循环的转换。处理这个问题的函数 `foreach()` 显示如下：

```
// Translate a foreach loop.
void foreach() {
    string token;
    string varname;
    string arrayname;

    char forvarname[5] = "_i";
    static char counter[2] = "a";

    // Create loop control variable for generated
    // for loop.
    strcat(forvarname, counter);
    counter[0]++;

    // Only 26 foreach loops in a file because
    // generated loop control variables limited to
    // _ia to _iz. This can be changed if desired.
    if(counter[0] > 'z')
        throw SyntaxExc("Too many foreach loops.");

    fout << "int " << forvarname
        << " = 0;\n";

    // Write beginning of generated for loop.
    fout << indent << "for(";

    skipspaces();

    // Read the (
    gettoken(token);
    if(token[0] != '(')
        throw SyntaxExc("( expected in foreach.");

    skipspaces();

    // Get the type of the foreach variable.
    gettoken(token);
    fout << token << " ";

    skipspaces();

    // Read and save the foreach variable's name.
    gettoken(token);
    varname = token;

    skipspaces();

    // Read the "in"
```

```

    gettoken(token);
    if(token != "in")
        throw SyntaxExc("in expected in foreach.");

    skipspaces();

    // Read the array name.
    gettoken(token);
    arrayname = token;

    fout << varname << " = " << arrayname << "[0];\n";

    // Construct target value.
    fout << indent + " " << forvarname << " < "
        << "((sizeof " << token << ")/"
        << "(sizeof " << token << "[0]));\n";

    fout << indent + " " << forvarname << "++,"
        << varname << " = " << arrayname << "["
        << forvarname << "]);";

    skipspaces();

    // Read the )
    gettoken(token);
    if(token[0] != ')')
        throw SyntaxExc(") expected in foreach.");
}

```

这个译码器将 `foreach` 循环转换为等价的 `for` 循环。这涉及到两个相当复杂的问题的处理。为了理解它们，考虑下面的示例：

```

double nums[] = { 1.1, 2.2, 3.3 };
double sum = 0.0;

foreach(double v in nums)
    sum += v;

```

`foreach` 循环被转换为下面的 C++ 代码：

```

int _ia = 0;
for(double v = nums[0];
    _ia < ((sizeof nums)/(sizeof nums[0]));
    _ia++, v = nums[_ia])
    sum += v;

```

首先，注意 `for` 循环需要一个循环控制变量，在这里称之为 `_ia`，但是在 `foreach` 语句中没有这种变量。记住，在 `foreach` 内声明的变量(在此情况下为 `v`)接收数组元素的值。它不是循环计数器。这意味着必须创建循环控制变量。这个变量不能在 `for` 循环内声明，因为这个声明必须为 `foreach` 变量 `v` 的声明保存，`v` 必须是循环内的局部变量。因此，必须在 `for` 循环外声明这个循环控制变量。从而要求生成一个惟一的整型变量名称，这个名称不与这个作用域内使用的任

何其他变量冲突。

其次, for 必须遍历数组中(在这里为 `nums`)的所有元素, 但是数组的大小不是 `foreach` 说明的一部分。这意味着必须计算数组中元素的个数。幸运的是, 可以使用 `sizeof` 完成此任务。

现在让我们浏览转换过程中的每一个步骤。`foreach` 转换的开始创建了一个(希望如此)惟一的变量, 这个变量将用作 `for` 循环控制变量。这个变量的名称以 `_i` 开始, 随后是 `a` 到 `z` 之间的一个字母。在文件中创建的第一个变量的名称为 `_ia`, 第二个为 `_ib`, 等。总共有 26 个这样的变量。这并不是生成无冲突变量名称的最好方法, 但是它的优点是简单, 对于试验 `foreach` 循环来说已经足够了。这个变量的类型为 `int`, 其声明写入到输出文件。

随后, 向输出文件写入 “`for(`”, `for` 循环由此开始。然后创建 `for` 循环的初始化部分。首先读取 `foreach` 变量的类型和名称, 并将其复制到输出文件。随后, 读取关键字 `in`, 然后将其丢弃。再读取数组的名称, 并用它来创建初始化部分, 将 `foreach` 变量的值设置为数组第一个元素的值。

随后生成 `for` 循环的条件部分。在这个部分, 循环控制变量与代表了数组元素数量的值比较。这个值是使用数组的大小除以数组中单个元素的大小来获得的。这个除法是必须的, 因为 `sizeof` 返回数组的字节数, 而不是数组中元素的数量。

最后, 创建 `for` 循环的迭代部分。它增加循环控制变量, 并且将 `foreach` 变量的值设置为数组下一个元素的值。在返回之前, `foreach()` 确定 `foreach` 语句以 “`)`” 结尾。

最后一点: 在 C# 中, `foreach` 可以用于数组和集合(类似于 STL 容器)。为了简单起见, 这个版本只能用于数组。然而, 您可以试着修改它, 使其能够用于遍历 STL 容器。

4.5.5 转换 `cases` 语句

如前所述, `cases` 语句提供了用一条语句取代一系列堆叠的 `case` 语句的方法。它是由这里给出的 `cases()` 函数转换的:

```
// Translate a cases statement.
void cases() {
    string token;
    int start, end;

    skipspaces();

    // Get starting value.
    gettoken(token);

    if(isdigit(token[0])) {
        // is an int constant
        start = atoi(token.c_str());
    }
    else if(token[0] == '\\') {
        // is char constant
        gettoken(token);

        start = (int) token[0];

        // discard closing '
    }
}
```

```

    gettoken(token);
    if(token[0] != '\\')
        throw SyntaxExc("' expected in cases.");
}
else
    throw SyntaxExc("Constant expected in cases.");

skipspaces();

// Read and discard the "to".
gettoken(token);
if(token != "to")
    throw SyntaxExc("to expected in cases.");

skipspaces();

// Get ending value.
gettoken(token);

if(isdigit(token[0])) {
    // is an int constant
    end = atoi(token.c_str());
}
else if(token[0] == '\\') {
    // is char constant
    gettoken(token);

    end = (int) token[0];

    // discard closing '
    gettoken(token);
    if(token[0] != '\\')
        throw SyntaxExc("' expected in cases.");
}
else
    throw SyntaxExc("Constant expected in cases.");

skipspaces();

// Read and discard the :
gettoken(token);

if(token != ":")
    throw SyntaxExc(": expected in cases.");

// Generate stacked case statements.
fout << "case " << start << ":\n";
for(int i = start+1 ; i <= end; i++) {
    fout << indent << "case " << i << ":";
    if(i != end) fout << endl;
}
}

```

尽管看起来代码很多，但实际上这只是一个简单的转换。`cases` 语句可以让您指定一个与 `switch` 表达式匹配的值的范围。译码器只是将这个范围转换为一系列堆叠的 `case` 语句。例如，这条 `switch` 语句：

```
switch(val) {  
    cases 0 to 3:  
        cout << "val is 0, 1, 2, or 3\n";  
        break;  
    case 4:  
        cout << "val is 4\n";  
}
```

被转换为等价的 C++ 代码：

```
switch(val) {  
    case 0:  
    case 1:  
    case 2:  
    case 3:  
        cout << "val is 0, 1, 2, or 3\n";  
        break;  
    case 4:  
        cout << "val is 4\n";  
}
```

正如您看到的那样，在此生成了代表 `cases` 语句指定的值范围的单个 `case` 语句。

`cases()` 函数是这样运行的。首先，读取这个范围的起始值。这个值或者是整数，或者是字符常量。如果是整型常量，则以数字开始。如果是字符常量，则以单引号开始。这个值作为整型值存储在 `start` 中。随后，读取并丢弃 `to`。然后读取结束值，它也是一个整型值或者字符常量。它的值存储在整型变量 `end` 中。最后，输出了一系列堆叠的 `case` 语句，每一条语句对应这个范围中的一个值。

4.5.6 转换 `typeof` 运算符

`typeof` 运算符支持另一种比较两种类型的方法。如前所述，其一般的形式为：

```
typeof op1 same as op2
```

在此，`op1` 和 `op2` 可以是标识符(如 `int` 或者 `MyClass`)或者对象。因此，`typeof` 可以用来比较两个对象的类型、一个对象的类型与已知类型，或者两个类型。如果两个类型相同，则返回 `true`；否则返回 `false`。`typeof` 运算符被转换为 C++ 表达式，来比较对每个操作数都应用了 `typeid` 运算符后得到的结果。

`typeof` 的转换由 `typeid()` 函数来处理，如下所示：

```
// Translate a typeof expression.  
void typeid() {  
    string token;  
    string temp;  
  
    fout << "typeid(";
```

```

    skipspaces();

    gettoken(token);

    do {
        temp = token;

        if(!gettoken(token))
            throw SyntaxExc("Unexpected EOF encountered.");

        if(token != "same") fout << temp;
    } while(token != "same");

    skipspaces();

    gettoken(token);

    if(token != "as") throw SyntaxExc("as expected.");

    fout << ")" == typeid(";

    skipspaces();

    do {
        if(!gettoken(token))
            throw SyntaxExc("Unexpected EOF encountered.");

        fout << token;
    } while(token != ")");
    fout << ")";
}

```

`typeid`的操作很容易跟踪。它读取第一个操作数，并输出包含了这个类型的 `typeid` 表达式。然后读取 `same` 和 `as` 关键字，并将其舍弃。随后读取第二个操作数，并输出相应的 `typeid` 表达式。因此，`typeid` 语句：

```

if(typeof obj same as MyClass)
    cout << "obj is a MyClass object.\n";

```

被转换为下面的 C++ 代码：

```

if(typeid(obj) == typeid(MyClass))
    cout << "obj is a MyClass object.\n";

```

4.5.7 转换 repeat/until 循环

如前所述，用译码器处理 `repeat/until` 循环主要是为了说明问题，因为很容易使用宏处理这个问题。然而，让译码器执行这个转换确实节省了您的精力，使得您不需要在每个使用它们的程序中都定义 `repeat` 和 `until` 宏。它还可以作为一个模型，可以对其进行修改来适应其他类型的循环。

使用 repeat() 方法转换关键字 repeat，如下所示，它简单地被替换为 do：

```
// Translate a repeat loop.
void repeat() {
    fout << "do";
}
```

关键字 until 由 until() 来处理，如下所示：

```
// Translate an until.
void until() {
    string token;
    int parencount = 1;

    fout << "while";

    skipspaces();

    // Read and store the {
    gettoken(token);
    if(token != "(")
        throw SyntaxExc("{ expected in typeof.");
    fout << "(";

    // Begin while by reversing and
    // parenthesizing the condition.
    fout << "!(";

    // Now, read the expression.
    do {
        if(!gettoken(token))
            throw SyntaxExc("Unexpected EOF encountered.");

        if(token == "(") parencount++;
        if(token == ")") parencount--;

        fout << token;
    } while(parencount > 0);
    fout << ")";
}
```

until() 函数用关键字 while 取代 until，然后反转条件表达式。（记住，repeat/until 循环在条件为 true 时终止。do/while 循环在条件为 false 时终止）。

尽管在概念上很简单，但在译码器的这个部分反转条件表达式还是需要花费一点力气。原因是不能简单地在表达式的开始加“!”。另外，必须将“!”开头的表达式加上括号。为了理解这样做的原因，考虑下面的 repeat/until 循环：

```
int i=0;

repeat {
    cout << i << " ";
```



```
    i-+;
} until (i==10);
```

它被转换为如下的 do/while 循环:

```
int i=0;

do {
    cout << i << " ";
    i++;
} while(!(i==10));
```

while 条件表达式中的括号确保 i 不等于 10 的时候循环一直运行。然而, 如果去掉括号, 如下所示:

```
}while(!i==10);
```

循环在 “!i” 等于 10 时运行, 这是一个完全不同的条件。

为了给 until 表达式加括号, 译码器必须在表达式的开头加入左括号, 在结尾加入右括号。问题是, 译码器如何知道条件表达式何时结束呢? 它通过对右括号计数做出判断。until 中的表达式包含在一对括号中。括号的计数存储在 parencount 变量中, 这个变量的初始值为 1(对应于 until 表达式的左括号)。当复制条件表达式时, 每发现一次左括号就增加这个计数, 每发现一次右括号就减小这个计数。因此, 当 parencount 为 0 时, 就到达了表达式的结尾, 就可以添加最后的右括号了。

4.6 演示程序

下面的程序演示了译码器支持的实验性功能。

```
// Demonstrate all of the experimental features
// supported by the translator.
#include <iostream>
using namespace std;

// Create a polymorphic base class.
class A {
public:
    virtual void f() { };
};

// And a concrete subclass.
class B: public A {
public:
    void f() {}
};

int main() {
    int n[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    double dn[] = {1.1, 2.2, 3.3, 4.4 };
```

```

cout << "Using a foreach loop.\n";

/* Keywords, such as foreach or typeof
   are ignored when inside comments
   or quoted strings. */

// A foreach loop.
foreach(int x in n )
    cout << x << ' ';

cout << "\n\n";

cout << "Using nested foreach loops.\n";

// A foreach loop with a block.
foreach(double f in dn) {
    cout << f << ' ';
    cout << f*f << ' ';

    // A nested foreach loop.
    foreach(double f in dn)
        cout << f/3 << " ";

    cout << endl;
}

cout << endl;

cout << "Demonstrate cases statement.\n";

cout << "A cases statement that uses integer constants:\n";

// Demonstrate cases statement that uses
// integer constants.
for(int i=0; i < 12; i++)
    switch(i) {
        case 0:
            cout << "case 0\n";
            break;
        cases 1 to 6:
            cout << "cases 1 to 6\n";
            break;
        case 7:
            cout << "case 7\n";
            break;
        cases 8 to 10:
            cout << "cases 8 to 10\n";
            break;
        default:
            cout << "case 11\n";
    }
}

```

```

cout << "\n";

cout << "A cases statement that uses character constants:\n";

// Demonstrate a cases statement that uses
// character constants.
for(char ch='a'; ch <= 'e'; ch++)
    switch(ch) {
        case 'a':
            cout << "case a\n";
            break;
        cases 'b' to 'd':
            cout << "cases b to d\n";
            break;
        case 'e':
            cout << "case e\n";
    }

cout << endl;

cout << "A repeat/until loop.\n";

// Demonstrate a repeat/until loop.
int k = 0;
repeat {
    k++;
    cout << "k: " << k << " ";
} until(k==10);

cout << "\n\n";

cout << "Use typeof.\n";

// Demonstrate typeof.
A *aPtr;
B *bPtr, bObj;

// Assign a base pointer to derived object.
aPtr = &bObj;
bPtr = &bObj;

if(typeof *aPtr same as *bPtr)
    cout << "aPtr points to same type of object as bPtr\n";

if(typeof *aPtr same as B)
    cout << "aPtr points to B object\n";

return 0;
}

```

当此程序通过译码器运行时，就生成了如下的 C++ 代码：

```
// Translated from an .exp source file.
// Demonstrate all of the experimental features
// supported by the translator.
#include <iostream>
using namespace std;

// Create a polymorphic base class.
class A {
public:
    virtual void f() { };
};

// And a concrete subclass.
class B: public A {
public:
    void f() {}
};

int main() {
    int n[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    double dn[] = {1.1, 2.2, 3.3, 4.4 };

    cout << "Using a foreach loop.\n";

    /* Keywords, such as foreach or typeof
       are ignored when inside comments
       or quoted strings. */

    // A foreach loop.
    int _ia = 0;
    for(int x = n[0];
        _ia < ((sizeof n)/(sizeof n[0]));
        _ia++, x = n[_ia])
        cout << x << ' ';

    cout << "\n\n";

    cout << "Using nested foreach loops.\n";

    // A foreach loop with a block.
    int _ib = 0;
    for(double f = dn[0];
        _ib < ((sizeof dn)/(sizeof dn[0]));
        _ib++, f = dn[_ib]) {
        cout << f << ' ';
        cout << f*f << ' ';

        // A nested foreach loop.
        int _ic = 0;
        for(double f = dn[0];
            _ic < ((sizeof dn)/(sizeof dn[0]));
            _ic++, f = dn[_ic])
```

```

        cout << f/3 << " ";

    cout << endl;
}

cout << endl;

cout << "Demonstrate cases statement.\n";

cout << "A cases statement that uses integer constants:\n";

// Demonstrate cases statement that uses
// integer constants.
for(int i=0; i < 12; i++)
    switch(i) {
        case 0:
            cout << "case 0\n";
            break;
        case 1:
        case 2:
        case 3:
        case 4:
        case 5:
        case 6:
            cout << "cases 1 to 6\n";
            break;
        case 7:
            cout << "case 7\n";
            break;
        case 8:
        case 9:
        case 10:
            cout << "cases 8 to 10\n";
            break;
        default:
            cout << "case 11\n";
    }

cout << "\n";

cout << "A cases statement that uses character constants:\n";

// Demonstrate a cases statement that uses
// character constants.
for(char ch='a'; ch <= 'e'; ch++)
    switch(ch) {
        case 'a':
            cout << "case a\n";
            break;
        case 98:
        case 99:
        case 100:

```

```

        cout << "cases b to d\n";
        break;
    case 'e':
        cout << "case e\n";
    }

    cout << endl;

    cout << "A repeat/until loop.\n";

    // Demonstrate a repeat/until loop.
    int k = 0;
    do {
        k++;
        cout << "k: " << k << " ";
    } while(!(k==10));

    cout << "\n\n";

    cout << "Use typeof.\n";

    // Demonstrate typeof.
    A *aPtr;
    B *bPtr, bObj;

    // Assign a base pointer to derived object.
    aPtr = &bObj;
    bPtr = &bObj;

    if(typeid(*aPtr) == typeid(*bPtr))
        cout << "aPtr points to same type of object as bPtr\n";

    if(typeid(*aPtr) == typeid(B))
        cout << "aPtr points to B object\n";

    return 0;
}

```

这个 C++ 版本可以被任何流行的 C++ 编译器编译，并生成了如下的输出：

```

Using a foreach loop.
1 2 3 4 5 6 7 8 9 10

```

```

Using nested foreach loops.
1.1 1.21 0.366667 0.733333 1.1 1.46667
2.2 4.84 0.366667 0.733333 1.1 1.46667
3.3 10.89 0.366667 0.733333 1.1 1.46667
4.4 19.36 0.366667 0.733333 1.1 1.46667

```

```

Demonstrate cases statement.
A cases statement that uses integer constants:
case 0

```

```

cases 1 to 6
cases 1 to 6
cases 1 to 6
cases 1 to 6
cases 1 to 6
cases 1 to 6
case 7
cases 8 to 10
cases 8 to 10
cases 8 to 10
case 11

```

A cases statement that uses character constants:

```

case a
cases b to d
cases b to d
cases b to d
case e

```

A repeat/until loop.

```
k: 1 k: 2 k: 3 k: 4 k: 5 k: 6 k: 7 k: 8 k: 9 k: 10
```

Use typeof.

```
aPtr points to same type of object as bPtr
```

```
aPtr points to B object
```

4.7 尝试完成以下任务

正如本章开始所讲述的那样，译码器的目的是为了您能够按照自己的想法实现新的语言特征或者有趣地改变现有特征。为了在译码器中加入您自己的实验性关键字，首先要创建一个函数来处理这个转换。然后，在主循环内加入另外的 else if 语句，当遇到新的关键字时，就会调用这个函数。

您可以试验任何您想到的结构。下面的想法可以帮助您开始：

- 一条 breakon 语句，当一些条件为 true 时打断一个循环。它可能使用这样的语法：

```
breakon(x==99);
```

- 一条 breakto 语句，将控制转移到循环之外，或者跳转到指定的标记。它可能使用这样的语法：

```
breakto jmp12;
```

- 一条 ignore 语句，当碰到某个值时，它引起循环的提前迭代。因此，它改进了 if...continue 语句。它可能使用这样的语法：

```
ignore(n==12);
```

当然，并不是所有的实验性功能都具有好的结果。

最后一点：如果您有兴趣设计自己的语言，就会发现第 9 章特别有价值，因为在那里为 C++ 的一个小子集开发了一个解释程序。您可以将这个解释程序作为起点，来寻求能够轻易处理比本章创建的译码器更为高级的语言特性。

第 5 章 Internet 文件下载工具

Internet 不可逆转地改变了程序设计的过程。在 Internet 出现之前,大多数应用程序都在单独的机器上孤立地执行,或者使用小型的局域网。Internet 改变了这种状况。现在,大多数计算机都连接到 Internet,许多程序都使用了万维网的大量资源。对于现代程序员,将 Internet 功能整合到应用程序中不再是一种选择,而是一种必然。

尽管 Internet 很重要,但是 C++ 并没有提供对其内建的支持。这是因为 C++ 在 20 世纪 90 年代后期 Internet 出现的时候就已经成熟了。然而这并非是一个障碍,事实上却是一个优点。C++ 允许访问操作系统的 Internet 功能,而不是指定了所有程序员都必须使用的惟一方法。通过让您选择到 Internet 最好的接口,不仅提供了潜在的高效率和灵活性,而且可以让您以最适合底层执行环境的方法来编写具有 Internet 功能的程序。因此,如果您想要建立高性能的、具有 Internet 功能的代码,C++ 是您的第一选择。

为了说明 C++ 可以轻易地处理 Internet 任务,本章开发了一个文件下载系统,它可以被整合到许多不同的基于 Internet 的应用程序中。当给定文件的 URL 时,这个文件下载工具从 Internet 获取这个文件。这是一个独立的子系统,可以处理打开 Internet 连接、读取文件以及随后的关闭连接中所有的细节。这个文件下载工具有一个特殊的特征:它可以断点续传。例如,如果在下载的过程中,与 Internet 的连接丢失,当连接重新建立时,这个下载可以在中断的那个点重新启动。当通过慢速的调制解调器连接下载很大的文件时,这个特性尤其重要。

在开发这个下载子系统之后,建立了两个使用它的应用程序。第一个是简单的、基于控制台的应用程序,只是为了演示这个下载工具。第二个是可以用来下载文件的 GUI 应用程序。

由于 Internet 支持是由操作系统提供的,因此有必要选择一个操作系统。对几乎所有本书的读者都适用的操作系统是 Windows,在本章就使用它。然而,基本的技术也可以应用到其他环境。

提示:

本章假定读者对 Internet 具有一般的理解,并且具有 Windows 程序设计的工作经验。对这两个问题的讲述远远超出了本书的范围。

5.1 WinINet 库

为了访问 Internet,本章使用了 Windows 提供的易于使用的库。这个库称为 Windows Internet,或者简称 WinINet。WinINet API 提供了多种高级的函数,以一致的、稳定的方式处理各种协议,如 HTTP 和 FTP。Windows 为您处理底层细节(在某种意义上,这类似于<fstream>为文件操作提供一致的接口,从而为您处理细节)。正如您所看到的,如果您遵循少许规则,就可以向任何 Windows 应用程序加入对 Internet 的访问。

尽管 WinINet 是一个很大的库，但是您只需要使用下面一些函数：

InternetAttemptConnect	检查 Internet 连接是否有效
InternetOpen	打开 Internet 连接并返回一个句柄
InternetOpenUrl	打开 URL 并返回一个句柄
HttpQueryInfo	从最后一个响应报头获取信息
InternetReadFile	从开放的 URL 读取字节
InternetCloseHandle	关闭 Internet 句柄

在讨论文件下载工具的代码时，会详细讨论这些函数。

为了使用 WinINet，必须在您的程序中包含 wininet.h，并且将您的应用程序与 wininet.lib 相链接。

5.2 文件下载工具子系统

文件下载工具子系统的整个代码如下所示。代码在文件 dl.cpp 中。

```
// A file download subsystem.
#include <iostream>
#include <windows.h>
#include <wininet.h>
#include <fstream>
#include <cstdio>

using namespace std;

const int MAX_ERRMSG_SIZE = 80;
const int MAX_FILENAME_SIZE = 512;
const int BUF_SIZE = 1024;

// Exception class for download errors.
class DLExc {
    char err[MAX_ERRMSG_SIZE];
public:
    DLExc(char *exc) {
        if(strlen(exc) < MAX_ERRMSG_SIZE)
            strcpy(err, exc);
    }

    // Return a pointer to the error message.
    const char * geterr() {
        return err;
    }
};

// A class for downloading files from the Internet.
class Download {
    static bool ishttp(char *url);
    static bool httpverOK(HINTERNET hUrl);
    static bool getfname(char *url, char *fname);
    static unsigned long openfile(char *url, bool reload,
```

```

                                ofstream &fout);

public:
    static bool download(char *url, bool restart=false,
        void (*update)(unsigned long, unsigned long)=NULL);
};

// Download a file.
//
// Pass the URL of the file to url.
//
// To reload a file, pass true to reload.
//
// To specify an update function that is called after
// each buffer is read, pass a pointer to that
// function as the third parameter. If no update
// function is desired, then let the third parameter
// default to null.
bool Download::download(char *url, bool reload,
    void (*update)(unsigned long, unsigned long)) {

    ofstream fout;           // output stream
    unsigned char buf[BUF_SIZE]; // input buffer
    unsigned long numrcvcd;    // number of bytes read
    unsigned long filelen;     // length of file on disk
    HINTERNET hUrl, hInet;    // Internet handles
    unsigned long contentlen;  // length of content
    unsigned long len;         // length of contentlen
    unsigned long total = 0;   // running total of bytes received
    char header[80];          // holds Range header

    try {
        if(!ishttp(url))
            throw DLExc("Must be HTTP url.");

        // Open the file specified by url.
        // The open stream will be returned
        // in fout. If reload is true, then
        // any preexisting file will be truncated.
        // The length of any preexisting file (after
        // possible truncation) is returned.
        filelen = openfile(url, reload, fout);

        // See if Internet connection available.
        if(InternetAttemptConnect(0) != ERROR_SUCCESS)
            throw DLExc("Can't connect.");

        // Open Internet connection.
        hInet = InternetOpen("downloader",
            INTERNET_OPEN_TYPE_DIRECT,
            NULL, NULL, 0);
    }
}

```

```

if(hInet == NULL)
    throw DLExc("Can't open connection.");

// Construct header requesting range of data.
sprintf(header, "Range:bytes=%d-", filelen);

// Open the URL and request range.
hIurl = InternetOpenUrl(hInet, url,
    header, -1,
    INTERNET_FLAG_NO_CACHE_WRITE, 0);

if(hIurl == NULL) throw DLExc("Can't open url.");

// Confirm that HTTP/1.1 or greater is supported.
if(!httpverOK(hIurl))
    throw DLExc("HTTP/1.1 not supported.");

// Get content length.
len = sizeof contentlen;
if(!HttpQueryInfo(hIurl,
    HTTP_QUERY_CONTENT_LENGTH |
    HTTP_QUERY_FLAG_NUMBER,
    &contentlen, &len, NULL))
    throw DLExc("File or content length not found.");

// If existing file (if any) is not complete,
// then finish downloading.
if(filelen != contentlen && contentlen)
do {
    // Read a buffer of info.
    if(!InternetReadFile(hIurl, &buf,
        BUF_SIZE, &numrcvd))
        throw DLExc("Error occurred during download.");

    // Write buffer to disk.
    fout.write((const char *) buf, numrcvd);
    if(!fout.good())
        throw DLExc("Error writing file.");

    total += numrcvd; // update running total

    // Call update function, if specified.
    if(update && numrcvd > 0)
        update(contentlen+filelen, total+filelen);

} while(numrcvd > 0);
else
    if(update)
        update(filelen, filelen);
} catch(DLExc) {
    fout.close();
}

```

```

    InternetCloseHandle(hIurl);
    InternetCloseHandle(hInet);

    throw; // rethrow the exception for use by caller
}

fout.close();
InternetCloseHandle(hIurl);
InternetCloseHandle(hInet);

return true;
}

// Return true if HTTP version of 1.1 or greater.
bool Download::httpverOK(HINTERNET hIurl) {
    char str[80];
    unsigned long len = 79;

    // Get HTTP version.
    if(!HttpQueryInfo(hIurl, HTTP_QUERY_VERSION, &str, &len, NULL))
        return false;

    // First, check major version number.
    char *p = strchr(str, '/');
    p++;
    if(*p == '0') return false; // can't use HTTP 0.x

    // Now, find start of minor HTTP version number.
    p = strchr(str, '.');
    p++;

    // Convert to int.
    int minorVerNum = atoi(p);

    if(minorVerNum > 0) return true;
    return false;
}

// Extract the filename from the URL. Return false if
// the filename cannot be found.
bool Download::getfname(char *url, char *fname) {
    // Find last /.
    char *p = strrchr(url, '/');

    // Copy filename after the last /.
    if(p && (strlen(p) < MAX_FILENAME_SIZE)) {
        p++;
        strcpy(fname, p);
        return true;
    }
    else
        return false;
}

```

```

}

// Open the output file, initialize the output
// stream, and return the file's length. If
// reload is true, first truncate any preexisting
// file.
unsigned long Download::openfile(char *url,
                                bool reload,
                                ofstream &fout) {
    char fname[MAX_FILENAME_SIZE];

    if(!getfname(url, fname))
        throw DLExc("File name error.");

    if(!reload)
        fout.open(fname, ios::binary | ios::out |
                  ios::app | ios::ate);
    else
        fout.open(fname, ios::binary | ios::out |
                  ios::trunc);

    if(!fout)
        throw DLExc("Can't open output file.");

    // Get current file length.
    return fout.tellp();
}

// Confirm that the URL specifies HTTP.
bool Download::ishttp(char *url) {
    char str[5] = "";

    // Get first four characters from URL.
    strncpy(str, url, 4);

    // Convert to lowercase
    for(char *p=str; *p; p++) *p = tolower(*p);

    return !strcmp("http", str);
}

```

注意，这个文件定义了两个类。第一个类是 `DLExc`，这个类封装了下载工具抛出的异常。`DLExc` 的构造函数传递了一个指向字符串的指针来描述这个异常，并存储这个字符串。调用成员函数 `geterr()` 可以获得指向错误信息的指针。

第二个类是 `Download`，这个类处理文件的下载。注意，`Download` 类只包含静态函数。因此，将下载工具放入到一个类中，不仅是一种封装的手段，更是一种组织的技术。事实上，可以使用名字空间而不是类来达到这个目的。然而，使用类可以将几个函数设置为私有的，从而阻止了其他代码的使用。另外，使用类简化了以后新特性的扩展。

下面部分详细介绍 `Download` 的各个部分。

5.2.1 操作的一般理论

在检查 Download 的各个部分之前，理解它操作的一般理论是有帮助的。为了下载文件，这个文件的 URL 被传递给 download() 函数。如果在磁盘上不存在同名文件，则这个文件被全部下载。然而，如果找到同名的已下载的部分文件，就只会下载这个文件的剩余部分。正是下载文件剩余部分的能力使得续传被中断的下载成为可能。

为了执行部分下载，Download 依赖于 HTTP 1.1 版本(或者更高版本)提供的功能：Range header，这个功能允许下载某个范围的数据。因此，这个下载工具只能使用支持 HTTP 1.1 版本或者更高版本的 URL。正是 Range header 使得被中断的下载可以从中断的那个点重启。由于需要 HTTP 1.1 版本，这个下载工具只支持 HTTP 下载。

当涉及到 HTTP 请求时，报头是伴随着这个请求的信息。Range 报头是一个字符串，其通常的形式为：

```
Range: bytes=start-end
```

在此，start 指定了这个范围的开始，end 指定了结尾。如果没有 end，这个范围从开头到文件的结束。

Download 定义了一个公有函数 download()，这个函数负责下载文件。因此，它是下载的入口点。Download 定义了 4 个支持函数：

ishttp	确定 URL 是否指定一个 HTTP 请求
httpverOK	确定使用的是否为 HTTP 1.1 版本或更高版本
getfname	获取 URL 的文件名部分
openfile	打开文件并返回其长度，如果已存在部分下载，那么长度将大于 0

在 Download 内，这些支持函数被声明为 private。

5.2.2 download() 函数

download() 函数是 Download 定义的惟一的公有函数。它由用户代码调用来处理文件的下载。因此，为了下载文件，需要调用 download() 函数。由于它的重要性，我们对其逐行检查。它的开始代码为：

```
bool Download::download(char *url, bool reload,
void (*update)(unsigned long, unsigned long)) {
```

download() 函数具有 3 个参数。首先是 url，这是一个指向字符串的指针，这个字符串包含了文件完整的 URL(包括文件名)。文件名假定为 URL 中最后一个“/”后面的名称。例如，下面的 URL 中，文件名为 MyFile.dat：

```
http://www.SoeSite.com/SomeDir/MyFile.dat
```

记住，必须使用规范的形式指定完整的 URL，包括 http: //。

第二个参数 reload 判断是否再次下载已经下载过的某个文件。如果 reload 为 true，则会下载整个文件，而不管磁盘上是否存在这个文件(无论存在部分或者全部)。如果 reload 为 false，那么只下载文件的剩余部分(如果有的话)。因此，如果您想要获取一个已下载文件的新副本，就将 reload 设置为 true。

第三个参数是 `update`。这是一个函数指针，它作为用户代码了解下载过程的一种方法，被 `download()` 周期性地调用。如果这个参数为 `null`，则不会使用 `update` 函数。我们将在稍后讨论对 `update` 函数参数的需求。

随后，声明了如下的局部变量：

```
ofstream fout;           // output stream
unsigned char buf[BUF_SIZE]; // input buffer
unsigned long numrcvd;    // number of bytes read
unsigned long filelen;    // length of file on disk
HINTERNET hUrl, hInet;   // Internet handles
unsigned long contentlen; // length of content
unsigned long len;        // length of contentlen
unsigned long total = 0;  // running total of bytes received
char header[80];         // holds Range header
```

注意变量 `hUrl` 和 `HInet`。它们分别拥有到被下载的 URL 的句柄和到 Internet 连接的句柄。`WinInet API` 函数会用到其中的一个句柄。

`main` 函数以下面所示的代码开始。注意整个下载代码被包装到一个 `try/catch` 块中，用来处理通信和/或者文件错误。

```
try {
    if(!ishttp(url))
        throw DLExc("Must be HTTP url.");
```

`download()` 做的第一件事是调用 `ishttp()`，通过验证指定的 URL 是否以 “http” 开始来判断它否是一个 HTTP 请求。如前所述，这个下载工具只能处理 HTTP 请求。

随后，打开接收文件并获取它的长度。

```
// Open the file specified by url.
// The open stream will be returned
// in fout. If reload is true, then
// any preexisting file will be truncated.
// The length of any preexisting file (after
// possible truncation) is returned.
filelen = openfile(url, reload, fout);
```

正如注释所说明的那样，如果这个文件不存在或者被重新下载，那么删除已经存在的同名文件。在此情况下，`openfile()` 返回的文件长度为 0。否则，如果这个文件已经存在，就会返回它的长度。

记住，如果某个下载被中断，当这个下载重启时，部分文件已经存在。

文件打开之后，下面的代码建立与 Internet 的连接：

```
// See if Internet connection available.
if(InternetAttemptConnect(0) != ERROR_SUCCESS)
    throw DLExc("Can't connect.");

// Open Internet connection.
hInet = InternetOpen("downloader",
                    INTERNET_OPEN_TYPE_DIRECT,
```

```
NULL, NULL, 0);
```

```
if(hInet == NULL)
    throw DLExc("Can't open connection.");
```

首先, 通过调用 WinINet API 函数 `InternetAttemptConnect()` 来试图与 Internet 建立连接。如果可能建立连接, 则这个函数返回 `ERROR_SUCCESS`。这个函数只保留了一个参数, 并且这个参数必须为 0。如果计算机当前没有连接, 那么(取决于您的计算机设置)将会看到一个对话框, 询问您是否想要连接。如果没有可用的连接, `InternetAttemptConnect()` 返回一个错误, 从而导致异常抛出。

假定某个连接可用, 就会调用 `InernetOpen()` 打开这个连接。这是另一个 WinINet 函数, 原型如下:

```
HINTERNET InternetOpen(LPCTSTR agent, DWORD access,
                        LPCTSTR proxy, LPCTSTR proxopt,
                        DWORD options)
```

在此, *agent* 指定了应用程序的名称, *access* 指定了使用的访问类型。对于这个下载工具, 这个访问的类型为 `INTERNET_OPEN_TYPE_DIRECT`。当 *access* 为 `INTERNET_OPEN_TYPE_DIRECT` 时, *proxy* 和 *proxopt* 都不会被使用, 并且必须为 `null`。参数 *options* 指定了任意的选项, 下载工具不需要这些选项。函数返回一个打开连接的句柄。如果这个连接没有被打开, 则返回 `null`。

一旦打开连接, 就会生成下面的 Range 报头:

```
// Construct header requesting range of data.
sprintf(header, "Range:bytes=%d-", filelen);
```

将这个报头传递给如下所示的 WinINet 函数 `InternetOpenUrl()`, 这个函数打开 `url` 中指定的 URL:

```
// Open the URL and request range.
hIurl = InternetOpenUrl(hInet, url,
                        header, -1,
                        INTERNET_FLAG_NO_CACHE_WRITE, 0);

if(hIurl == NULL) throw DLExc("Can't open url.");
```

`InternetOpenUrl()` 函数的原型如下:

```
HINTERNET InternetOpenUrl(HINTERNET hInet, LPCTSTR url,
                           LPCTSTR headerstr, DWORD headlen,
                           DWORD options, LPDWORD extra);
```

`InternetOpen()` 获得的 Internet 句柄传递给 `hInet`。指向包含打开的 URL 的字符串的指针传递给 `url`。一个指向字符串的指针传递给了 `headerstr`, 这个字符串包含了一个或者多个将被整合到请求中的附加报头。`headerstr` 的长度传递给 `headlen`, 如果 `headerstr` 指向一个 null-terminated 字符串, 则它可以为 -1, 就如同下载工具那样。在 `option` 中可以指定各种标记。这个下载工具所使用的是:

INTERNET_FLAG_NO_CACHE_WRITE

这样做可防止缓存下载的文件。任何附加的、应用程序指定的信息都可以传递给 `extra`。由于这个下载工具不需要这些信息，因此使用 0。这个函数将返回开放 URL 的句柄，如果失败，则返回空值。

服务器的每个响应都带有一个报头。这个报头的组件之一包含 HTTP 的版本信息。由于这个下载工具要求服务器支持 HTTP 1.1 版本或者更高版本，那么在开始处理之前，有必要检查 HTTP 的版本。这个任务是通过调用 `httpverOK()` 来完成的，如下所示：

```
// Confirm that HTTP/1.1 or greater is supported.
if(!httpverOK(hIurl))
    throw DLExc("HTTP/1.1 not supported.");
```

假如 HTTP 的版本是可以接受的，则下一步将获取下载内容的数量。调用 `WinINet` 函数 `HttpQueryInfo()` 可以完成这个任务，如下所示：

```
// Get content length.
len = sizeof contentlen;
if(!HttpQueryInfo(hIurl,
    HTTP_QUERY_CONTENT_LENGTH |
    HTTP_QUERY_FLAG_NUMBER,
    &contentlen, &len, NULL))
    throw DLExc("File or content length not found.");
```

当 `HttpQueryInfo()` 返回时，内容的长度(按字节数)存储在 `contentlen` 中。内容长度自动考虑 `Ranger header` 请求的范围。因此，如果整个文件将被下载，则请求的范围将会是 "0-" (它指定整个文件)，内容的长度将与文件的长度相等。当续传一个被中断的下载时，范围值将指定文件中间的一个点作为起始点，内容的长度等于剩余的字节数。

`HttpQueryInfo()` 从报头获取信息。其原型如下：

```
BOOL HttpQueryInfo(HINTERNET hIurl, DWORD what, LPVOID buf,
    LPDWORD buflen, LPDWORD index)
```

在此情况下，将由 `InternetOpenUrl()` 返回的请求的句柄传递给 `hIurl`。将指定被获取的信息项的值传递给 `what`。对于这个下载工具，需要两个值。第一个值是

`HTTP_QUERY_CONTENT_LENGTH`

用来获取被请求的内容的长度(在此情况下是文件)，第二个值是

`HTTP_QUERY_FLAG_NUMBER`

它要求这个值以整型值表示。把指向接收信息的缓冲区的指针传递给 `buf`，指定缓冲区长度的整型值的指针传递给 `buflen`。参数 `index` 允许您指定报头的索引，但是这个下载工具不需要这个特征，因此传递 `null`。如果能够获取请求的信息，函数则返回 `true`。内容的长度有可能不可使用，在此情况下 `HttpQueryInfo()` 返回 `false`。

假定内容的长度可用，如果磁盘上现有文件的长度小于内容的长度，并且内容的长度不为 0，这个文件(或者文件的剩余部分)用下面的代码下载：

```

// If existing file (if any) is not complete,
// then finish downloading.
if(filelen != contentlen && contentlen)
do {
    // Read a buffer of info.
    if(!InternetReadFile(hIurl, &buf,
                        BUF_SIZE, &numrcvd))
        throw DLExc("Error occurred during download.");

    // Write buffer to disk.
    fout.write((const char *) buf, numrcvd);
    if(!fout.good())
        throw DLExc("Error writing file.");

    total += numrcvd; // update running total

    // Call update function, if specified.
    if(update && numrcvd > 0)
        update(contentlen+filelen, total+filelen);
} while(numrcvd > 0);
else {
    if(update)
        update(filelen, filelen);
}

```

这段代码使用了 WinINet 的函数 `InternetReadFile()` 来读取文件，每次读取一个缓冲区。每循环一次，都会调用 `update` 指向的函数，除非 `update` 是 `null`。如前所述，`update` 所指向的函数用来报告文件的下载进度。

`InternetReadFile()` 是一个非常有用的函数，因为它使得从 Internet 读取文件的方式与从磁盘读取文件的方式相同。其原型如下：

```

BOOL InternetReadFile(HINTERNET hIurl, LPVOID buf, DWORD numbytes,
                    LPDWORD numrcvd);

```

文件的句柄传递给 `hIurl`。对于这个下载工具，这是由 `InternetOpenUrl()` 返回的句柄。将指向接收数据的缓冲区的指针传递给 `buf`，将要读取的字节数传递给 `numbytes`。这个值不能超过 `buf` 的长度。实际读取的字节数返回到 `numrcvd` 所指的变量中。当不再需要获取字节时，这个值为 0。如果成功，这个函数返回 `true`，否则返回 `false`。

`download()` 函数以下的代码结束：

```

} catch(DLExc) {
    fout.close();
    InternetCloseHandle(hIurl);
    InternetCloseHandle(hInet);

    throw; // rethrow the exception for use by caller
}

fout.close();
InternetCloseHandle(hIurl);
InternetCloseHandle(hInet);

```

```
    return true;
}
```

如果成功地完成，则 `download()` 函数将通过关闭文件和 Internet 句柄结束，并返回 `true`。如果发生错误，则仍然会关闭句柄，并抛出异常。这样做允许用户代码对错误做出响应。

5.2.3 ishttp()函数

如前所述，这个下载工具只支持 HTTP 文件下载。Download 使用这里所示的 `ishttp()` 函数来确定文件的 URL 指定了 HTTP 协议：

```
// Confirm that the URL specifies HTTP.
bool Download::ishttp(char *url) {
    char str[5] = "";

    // Get first four characters from URL.
    strncpy(str, url, 4);

    // Convert to lowercase
    for(char *p=str; *p; p++) *p = tolower(*p);

    return !strcmp("http", str);
}
```

`ishttp()` 的操作相当直接。它只是检查 URL 中前四个字符是否包含了字符串“http”。如果确实如此，它返回 `true`，否则返回 `false`。

5.2.4 httpverOK()函数

如下所示的 `httpverOK()` 函数确定处理请求的服务器是否支持 HTTP 1.1 版本：

```
// Return true if HTTP version of 1.1 or greater.
bool Download::httpverOK(HINTERNET hUrl) {
    char str[80];
    unsigned long len = 79;

    // Get HTTP version.
    if(!HttpQueryInfo(hUrl, HTTP_QUERY_VERSION, &str, &len, NULL))
        return false;

    // First, check major version number.
    char *p = strchr(str, '/');
    p++;
    if(*p == '0') return false; // can't use HTTP 0.x

    // Now, find start of minor HTTP version number.
    p = strchr(str, '.');
    p++;

    // Convert to int.
    int minorVerNum = atoi(p);
```

```

    if(minorVerNum > 0) return true;
    return false;
}

```

URL 的句柄传递给 `httpverOK()`。随后，通过对这个句柄调用 `HttpQueryInfo()`，并指定 `HTTP_QUERY_VERSION`（它请求版本信息），获取了 HTTP 的版本（以字符串的形式）。这个查询将版本字符串保存在 `str` 中。对于 HTTP 1.1 版本，字符串的形式如下：

```
HTTP/1.1
```

然后，这个函数调用标准库函数 `strchr()` 将 `p` 指向字符 “/”。接着增加 `p`，并确保它所指的数字不为 0。下一步，建立一个指向句号的指针来分割主版本号和次版本号。然后增加这个指针，使它指向句号后数值的第一个数字。最后，通过使用标准库中的 `atoi()` 函数将这个值转换为整型值。如果次版本号大于等于 1，那么 HTTP 版本大于等于 1.1。

5.2.5 getfname()函数

如下所示的 `getfname()` 函数从 URL 中提取文件名：

```

// Extract the filename from the URL. Return false if
// the filename cannot be found.
bool Download::getfname(char *url, char *fname) {
    // Find last /.
    char *p = strrchr(url, '/');

    // Copy filename after the last /.
    if(p && (strlen(p) < MAX_FILENAME_SIZE)) {
        p++;
        strcpy(fname, p);
        return true;
    }
    else
        return false;
}

```

在函数返回时，这个函数传递了一个指向保存 URL 的字符串的指针，以及一个指向保存了文件名的字符数组的指针。文件名假定为 URL 中最后一个 “/” 到结尾之间的部分。如果成功，则这个函数返回 `true`；如果找不到文件名，则返回 `false`。

5.2.6 openfile()函数

`openfile()` 函数打开一个磁盘文件，下载的文件将写入到这个文件中。它还返回现有文件的长度。如下所示：

```

// Open the output file, initialize the output
// stream, and return the file's length. If
// reload is true, first truncate any existing
// file.

```

```

unsigned long Download::openfile(char *url,
                                bool reload,
                                ofstream &fout) {
    char fname[MAX_FILENAME_SIZE];

    if(!getfname(url, fname))
        throw DLExc("File name error.");

    if(!reload)
        fout.open(fname, ios::binary | ios::out |
                  ios::app | ios::ate);
    else
        fout.open(fname, ios::binary | ios::out |
                  ios::trunc);

    if(!fout)
        throw DLExc("Can't open output file.");

    // Get current file length.
    return fout.tellp();
}

```

openfile()函数有 3 个参数：**url**、**reload** 和 **fout**。将指向包含 URL 的字符串的指针传递给 **url**。判断是否删除任何相同名称的现有文件的值传递给 **reload**。传递给 **fout** 一个指针，这个指针指向一个变量，当 **openfile()**返回时，这个变量包含了打开的文件流。

首先，**openfile()**从 **url** 指向的字符串获取文件名，然后检查 **reload** 的值。如果 **reload** 为 **true**，则打开输出文件，同时任何现有文件的内容都被删除。如果 **reload** 为 **false**，则会打开文件，而不会删除现有文件的任何内容，并指定所有的输出都在文件的结尾发生。然后将文件指针移动到文件结尾。随后，通过调用 **tellp()**来获取文件的长度。对于新建的文件，长度为 0。然而，如果找到现有的文件，这个值就是现有文件的长度，因为当打开文件时，文件指针在文件的结尾处。返回文件的长度。

5.2.7 update()函数

如前所述，如果想要跟踪下载的进度，就必须向 **download()**的 **update** 参数传递一个指向函数的指针，这个函数将会接收跟踪的信息。(对于后台操作，**update** 可以使用默认的 **null**)。跟踪函数必须具有如下的原型。(当然，函数名可以不同)。

```
void update(unsigned long total, unsigned long part);
```

当调用 **update()**时，文件的总长度传递给 **total**，当前下载的数量传递给 **part**。可以使用这些信息来监视下载的进度，并让用户知道。

5.3 Download 头文件

任何使用这个下载工具的文件都必须包含如下的头文件。这个文件的名称为 `dl.h`。

```
// Header file for downloader. Call this file dl.h.
#include <iostream>
#include <string>
#include <windows.h>
#include <wininet.h>
#include <fstream>
using namespace std;

const int MAX_ERRMSG_SIZE = 80;

// Exception class for downloading errors.
class DLExc {
    char err[MAX_ERRMSG_SIZE];
public:
    DLExc(char *exc) {
        if(strlen(exc) < MAX_ERRMSG_SIZE)
            strcpy(err, exc);
    }

    const char * geterr() {
        return err;
    }
};

class Download {
    static bool httpverOK(HINTERNET hUrl);
    static bool getfname(char *url, char *fname);
    static unsigned long openfile(char *url, bool reload,
                                   ofstream &fout);
public:
    static bool download(char *url, bool restart=false,
                          void (*update)(unsigned long, unsigned long)=NULL);
};
```

5.4 文件下载工具的演示

下面的程序建立了一个简单的下载文件的控制台应用程序来演示这个文件下载工具。为了前后一致，称其为 `dltest.cpp`。

```
// A Sample program that uses Download.
#include <iostream>
#include "dl.h"

// This function displays the download progress as a percentage.
void showprogress(unsigned long total, unsigned long part) {
```

```

    int val = (int) ((double) part/total*100);
    cout << val << "%" << endl;
}

int main(int argc, char *argv[])
{
    // This URL is for demonstration purposes only. Substitute
    // the URL of the file that you want to download.
    char url[] =
        "http://www.osborne.com/products/0072226803/0072226803_code.zip";

    bool reload = false;

    if(argc==2 && !strcmp(argv[1], "reload"))
        reload = true;

    cout << "Beginning download.\n";

    try {
        if(Download::download(url, reload, showprogress))
            cout << "Download Complete\n";
    } catch(DLExc exc) {
        cout << exc.geterr() << endl;
        cout << "Download Interrupted\n";
    }

    return 0;
}

```

在这个程序中，有3件很有趣的事情。首先，要注意到它包含了一个硬编码的URL。这个URL指定了一个文件，这个文件包含了免费的、在线的本书作者著的另一本书(*C++: The Complete Reference, 4th Edition*)的代码。这个文件方便地测试了这个下载工具。当然，您可以指定自己选择的URL。

随后，注意 `showprogress()` 函数。指向这个函数的一个指针被传递给 `download()` 的 `update` 参数。这意味着每次下载一个数据缓冲区时都会调用此函数。显示了一个值来指示完成的百分比。

最后要注意，`dltest` 接收一个命令行参数。如果这个参数为“reload”，那么不管磁盘上是否存在这个文件，都会将其全部下载。如果没有给出这个参数，只有在磁盘上这个文件不完整时，这个文件(或者这个文件的剩余部分)才会被下载。

为了编译 `dltest.cpp`，必须链接 `wininet.lib`。例如，如果使用 Visual C++ 的命令行编译，就可以使用如下的命令行来编译这个程序：

```
cl -GX dl.cpp dltest.cpp wininet.lib
```

如果使用 IDE，就必须记得在链接中加入 `wininet.lib`。

5.5 基于 GUI 的下载工具

无论何时您需要从 Internet 下载程序代码内的文件,都可以使用 Download 类。例如,可以使用下载工具获取每周的销售报表。如果加入前端用户界面,它还可以用作独立的 Internet 实用工具。为了说明第二个用途,本章为 Windows 开发了一个完整的,基于 GUI 的文件下载工具,其名称为 WinDL。

WinDL 显示了一个对话框,允许用户输入将要下载的文件 URL。它有一个进度条来显示下载的状态。它还有一个复选框来让用户请求完全下载文件。图 5-1 显示了这个文件下载对话框。

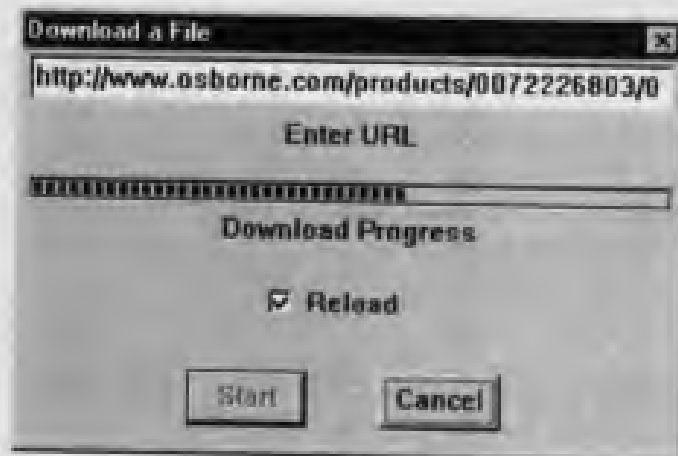


图 5-1 文件下载对话框

5.5.1 WinDL 代码

WinDL 由两个主要部分组成。首先当然是 Download 的代码,这在前面已经讲述过。第二个部分是处理 GUI 界面的代码。这些代码在下面给出。为了与前面一致,称这个文件为 windl.cpp。

```
// WinDL: A GUI-based file download utility.

#include <windows.h>
#include <commctrl.h>
#include <cstring>
#include <cstdio>
#include "windl.h"
#include <process.h>
#include "dl.h"

const int URL_BUF_SIZE = 1024;

LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);
BOOL CALLBACK DialogFunc(HWND, UINT, WPARAM, LPARAM);

void showprogress(unsigned long total,
                  unsigned long part);
```



```

void resetprogress();
unsigned __stdcall dlstart(void * reload);

char szWinName[] = "Download"; // name of window class

HINSTANCE hInst; // instance handle
HWND hwnd;      // handle of main window
HWND hProgWnd;  // handle of progress bar

HANDLE hThrd = 0; // thread handle
unsigned long Tid; // thread ID

// Progress counters.
int percentdone = 0;
int oldpercentdone = 0;

// A small struct for passing info to dlstart().
struct ThrdInfo {
    char *url; // pointer to URL string
    int reload; // reload flag
    HWND hPBStart; // handle of Start button
};

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                  LPSTR lpszArgs, int nWinMode)
{
    MSG msg;
    WNDCLASSEX wcl;
    INITCOMMONCONTROLSEX cc;

    // Define a window class.
    wcl.cbSize = sizeof(WNDCLASSEX);

    wcl.hInstance = hThisInst; // handle to this instance
    wcl.lpszClassName = szWinName; // window class name
    wcl.lpfnWndProc = WindowFunc; // window function
    wcl.style = 0; // default style

    wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); // large icon
    wcl.hIconSm = NULL; // use small version of large icon
    wcl.hCursor = LoadCursor(NULL, IDC_ARROW); // cursor style

    wcl.lpszMenuName = NULL; // no menu

    wcl.cbClsExtra = 0; // no extras
    wcl.cbWndExtra = 0;

    wcl.hbrBackground = NULL; // not used

    // Register the window class.
    if(!RegisterClassEx(&wcl)) return 0;

```

```

// Create a main window that won't be visible.
hwnd = CreateWindow(
    szWinName, // name of window class
    "File Downloader", // title
    0, // no style needed
    0, 0, 0, 0, // no dimensions
    NULL, // no parent window
    NULL, // no menu
    hThisInst, // instance handle
    NULL // no additional arguments
);

hInst = hThisInst; // save the current instance handle

// Initialize the common controls. This is
// needed because of the progress bar.
cc.dwSize = sizeof(INITCOMMONCONTROLSEX);
cc.dwICC = ICC_PROGRESS_CLASS;
InitCommonControlsEx(&cc);

// Show the window minimized.
ShowWindow(hwnd, SW_SHOWMINIMIZED);

// Create the download dialog box.
DialogBox(hInst, "DLDB", hwnd, (DLGPROC) DialogFunc);

// Create the message loop.
while(GetMessage(&msg, NULL, 0, 0))
{
    TranslateMessage(&msg); // translate keyboard messages
    DispatchMessage(&msg); // return control to Windows
}

return msg.wParam;
}

// Window function.
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                             WPARAM wParam, LPARAM lParam)
{
    switch(message) {
        case WM_DESTROY:
            PostQuitMessage(0); // terminate the program
            break;
        default:
            return DefWindowProc(hwnd, message, wParam, lParam);
    }
    return 0;
}

// Downloader Dialog function.
BOOL CALLBACK DialogFunc(HWND hwnd, UINT message,

```

```

                                WPARAM wParam, LPARAM lParam)
{
    // Here, url is initialized with a sample url for
    // demonstration purposes only.
    static char url[URL_BUF_SIZE] =
        "http://www.osborne.com/products/0072226803/0072226803_code.zip";

    static ThrdInfo ti;
    switch(message) {
        case WM_INITDIALOG:
            // Initialize edit box with URL.
            SetDlgItemText(hdwnd, IDD_EB1, url);

            // Create progress bar.
            hProgWnd = CreateWindow(PROGRESS_CLASS,
                                    "",
                                    WS_CHILD | WS_VISIBLE | WS_BORDER,
                                    4, 64, 320, 12,
                                    hdwnd, NULL, hInst, NULL);

            // Set step increment to 1.
            SendMessage(hProgWnd, PBM_SETSTEP, 1, 0);

            return 1;
        case WM_COMMAND:
            switch(LOWORD(wParam)) {
                case IDCANCEL:
                    EndDialog(hdwnd, 0);
                    PostQuitMessage(0);

                    return 1;

                case IDD_START: // start download
                    // Set position to zero.
                    SendMessage(hProgWnd, PBM_SETPOS, 0, 0);

                    // Get URL from edit box.
                    GetDlgItemText(hdwnd, IDD_EB1, url, URL_BUF_SIZE);
                    ti.url = url;

                    // Get reload status.
                    ti.reload = SendDlgItemMessage(hdwnd, IDD_CB1,
                                                    BM_GETCHECK, 0, 0);

                    // Get handle to Start button.
                    ti.hPBStart = GetDlgItem(hdwnd, IDD_START);

                    // Reset progress counters.
                    resetprogress();

                    // Start download thread.
                    if(!hThrd)

```

```

        hThrd = (HANDLE) _beginthreadex(NULL, 0, dlstart,
        (void *) &ti, 0, (unsigned *) &Tid);

        return 1;
    }
}

return 0;
}

// Show progress in the progress bar. This is called
// by the download() function.
void showprogress(unsigned long total,
                  unsigned long part) {

    percentdone = (part*100)/total;

    if(percentdone > oldpercentdone) {
        for(int i= oldpercentdone; i < percentdone; i++) {
            // Advance the progress bar.
            SendMessage(hProgWnd, PBM_STEPIT, 0, 0);
        }
        oldpercentdone = percentdone;
    }
}

// Reset the progress counters.
void resetprogress() {
    percentdone = 0;
    oldpercentdone = 0;
}

// Thread entry function that begins downloading.
unsigned __stdcall dlstart(void * param) {
    ThrdInfo *tip = (ThrdInfo *) param;

    // Disable Start button.
    EnableWindow(tip->hPBStart, 0);

    try {
        if(tip->reload == BST_CHECKED)
            Download::download(tip->url, true, showprogress);
        else
            Download::download(tip->url, false, showprogress);
    } catch(DLExc exc) {
        MessageBox(hwnd, exc.geterr(),
                  "Download Error", MB_OK);
    }

    // Enable Start button.
    EnableWindow(tip->hPBStart, 1);
}

```

```

    CloseHandle(hThrd); // close the thread handle
    hThrd = 0; // set thread handle to inactive

    return 0;
}

```

WinDL 使用下列源文件:

```

// Resources for file downloader.
#include <windows.h>
#include "windl.h"

DLDB DIALOGEX 18, 18, 164, 100
CAPTION "Download a File"
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION |
    WS_SYSMENU | WS_VISIBLE
{
    PUSHBUTTON "Start", IDD_START, 42, 80, 30, 14
    PUSHBUTTON "Cancel", IDCANCEL, 90, 80, 30, 14

    CTEXT "Download Progress", IDD_TEXT2, 2, 40, 160, 12

    CTEXT "Enter URL", IDD_TEXT1, 2, 16, 160, 12
    EDITTEXT IDD_EB1, 2, 1, 160, 12, ES_LEFT | WS_CHILD |
        WS_VISIBLE | WS_BORDER | ES_AUTOHSCROLL

    AUTOCHECKBOX "Reload", IDD_CB1, 62, 56, 36, 14
}

```

源文件和程序代码都需要包含头文件 `windl.h`, 如下所示:

```

#define IDD_START 100

#define IDD_CB1 200

#define IDD_EB1 300

#define IDD_TEXT1 401
#define IDD_TEXT2 402

```

为了编译 WinDL, 必须创建一个包含下面文件的项目:

```
dl.cpp    windl.cpp    windl.rc
```

头文件 `dl.h` 和 `windl.h` 也必须可用。您一定还要记得链接 `wininet.lib` 和 `comctl32.lib`。(进度控制需要 `comctl32.lib` 库)。最后, 由于 `download()` 在它自己的线程内运行, 因此必须链接多线程库。

5.5.2 WinDL 的运行方式

WinDL 为 Download 类提供了可视化的前端来处理用户的输入, 并显示下载的进度。WinDL 开始时建立了一个最小化的主窗口, 然后显示了下载对话框。因此, WinDL 是基于对话框的应

用程序，永远不会显示主窗口。如前所述，对于 WinDL 中支持所有 Windows 程序都通用的基本 Windows 框架部分的描述超出了本书的范围。然而，这些与下载工具有特殊关联的代码会在下面讲述。

处理用户与对话框交互的函数是 `DialogFunc()`。它声明了两个静态变量。第一个是名为 `url` 的数组，它为文本编辑框提供了支持数组。第二个是 `ti`，这是一个很小的结构，包含了将要传递给线程入口函数的信息。

当建立这个对话框时，初始化了文本编辑框和进度条。正如注释提到的那样，这个编辑框用 URL 来初始化只是为了演示(通常，此文本编辑框不会包含初始化字符串)。进度条的增量设置为 1。在默认情况下，进度条的范围为 0 到 100，因此当增量设置为 1 时，进度条的每次增加都是 1%。

为了下载文件，将所需文件的 URL 输入到文本编辑框中，然后单击 **Start** 按钮。这产生了如下的事件序列。首先，进度条设置为 0。然后从文本编辑框中获取包含了 URL 的字符串，从复选框中获取续传状态，并获取 **Start** 按钮的句柄。这些都存储在 `ti` 中，`ti` 将被传递给线程入口函数。随后，进度计数器被设置为 0。全局变量 `percentdone` 和 `oldpercentdone` 用来更新下载期间进度条的位置。最后，启动将处理下载的新线程。

必须在自己的线程中运行 `download()`，因为 Windows 消息传送系统假定控制权会相对快速地交还给 Windows。(也就是说，`DialogFunc()` 不能采取一些其他的操作阻止新消息的处理)。这个线程的入口函数为 `dlstart()`。正如第 3 章讲述的那样，在 Windows 中，所有的线程入口函数都只有一个 `void*` 参数。可以给这个参数传递函数需要的任何内容。在此情况下，传递了一个指向 `ti` 结构的指针，`ti` 包含了 3 个字段：`url`、`reload` 以及 `hPBStart`。`url` 字段指向用来下载的 URL，`Reload` 复选框的状态存储在 `reload` 字段中。它决定这个文件是否被完全重新下载。**Start** 按钮的句柄在 `hPBStart` 字段中。当开始下载时，`dlstart()` 用它将 **Start** 按钮变得不可用，当下载结束时，将其设置为可用。

`dlstart()` 函数调用 `download()` 来处理文件的下载。当这样做的时候，它向 `update` 参数传递 `showprogress` 的地址。当 `download()` 结束时，**Start** 按钮可用，线程句柄关闭。

`download()` 调用 `showprogress()` 函数，以显示下载的进度。它只是在文件的百分之一下载完成时，简单地增加进度条。

5.6 尝试完成以下任务

`Download` 可以添加几个增强的功能。首先您想要尝试的是加入下载给定 FTP 地址的文件的能力。由于 `InternetOpenUrl()` 支持 FTP，因此这个任务相对简单。尽管 HTTP 1.0 版本使用地越来越少，但还是想加入对它的支持。这也很容易，因为只需要总是下载全部的文件就可以了。另一个对下载工具有意义的增强是使得它能够获取一个文件的列表。做到这一点的方法之一是让它从磁盘文件读取 URL。最后，您可能想要添加自动重试功能，在下载被中断时自动尝试完成下载。

一般文件下载机制的应用功能不仅仅是基本的文件下载工具。`Download` 类可以在任何需要获取文件的时候使用。例如，您可以使用 `Download` 建立一个远程数据采集程序，从固定的站点下载数据文件，如一个库存报告单。

第 6 章 使用 C++ 的财务计算

尽管那些庞大而成熟的应用程序，如编译器、Internet 浏览器、文字处理器、数据库以及财务软件包等，占据了计算的主流，但是仍然存在一类普遍而简短的程序。这些程序执行不同的财务计算，如计算贷款的定期偿还、投资的预期价值或者贷款余额。这些计算都不复杂，也不需要很多的代码，然而它们产生的信息却非常有用。

由于 C++ 的特长在于建立高效的系统软件，因此对于财务应用程序，它经常被忽略，然而这是一个失误。C++ 在这个领域内有卓越的性能，它支持一整套数学函数和高效的浮点数运算。另外，由于它可以生成特别快速的执行代码，因此 C++ 非常适合于执行复杂的财务分析和建模的程序。

为了说明 C++ 能够轻松处理财务计算，本章开发了许多小程序来执行如下所示的财务计算：

- (1) 贷款的定期偿还
- (2) 投资的预期价值
- (3) 为了获得预期的价值所需的初始投资
- (4) 为了获得预期的养老金所需的投资
- (5) 投资所能得到的养老金的最大值
- (6) 贷款余额

这些程序可以直接使用，也可以将其修改以适应您的需要。坦白地说，尽管它们是这本书的最简单程序，但是它们可能也是您使用次数最多的程序。

6.1 计算贷款的定期偿还

最常用的财务计算可能就是计算贷款的定期偿还，如汽车或者住房贷款。贷款的偿还金额用下面的公式来计算：

$$\text{Payment} = (\text{intRate} * (\text{principal} / \text{payPerYear})) / (1 - ((\text{intRate} / \text{payPerYear}) + 1)^{\text{payPerYear} * \text{numYears}})$$

其中 `intRate` 为利率，`principal` 为本金，`payPerYear` 为每年偿还的数额，`numYears` 为贷款年限。

在下面的程序中，函数 `regpay()` 使用前面公式计算了贷款的偿还金额。本金、利率、贷款年限以及每年的偿还金额传递给这个函数。这个函数返回要偿还的金额。

```
#include <iostream>
#include <cmath>
#include <iomanip>
#include <locale>
```

```

using namespace std;

// Compute the regular payments on a loan.
double regpay(double principal, double intRate,
               int numYears, int payPerYear) {
    double numer;
    double denom;
    double b, e;

    intRate /= 100.0; // convert percentage to fraction

    numer = intRate * principal / payPerYear;

    e = -(payPerYear * numYears);
    b = (intRate / payPerYear) + 1.0;

    denom = 1.0 - pow(b, e);

    return numer / denom;
}

int main() {
    double p, r;
    int y, ppy;

    // Set locale to English. Adjust as necessary
    // for your language and/or region.
    cout.imbue(locale("english"));

    cout << "Enter principal: ";
    cin >> p;

    cout << "Enter interest rate (as a percentage): ";
    cin >> r;

    cout << "Enter number years: ";
    cin >> y;

    cout << "Enter number of payments per year: ";
    cin >> ppy;

    cout << "\nPayment: " << fixed << setprecision(2)
          << regpay(p, r, y, ppy) << endl;

    return 0;
}

```

为了计算贷款的偿还金额，只需要按提示输入所需的信息。下面是一个运行的样本：

```

Enter principal: 1000
Enter interest rate (as a percentage): 9
Enter number years: 5

```



```
Enter number of payments per year: 12
```

```
Payment: 20.76
```

在 `main()` 函数中，有两点值得注意。首先，将与 `cout` 相关的 `locale` 设置为英语。这是通过调用成员函数 `imbue()` 并向 `locale` 对象传递 `English` 来完成的。这意味着货币值将使用英语的习惯来显示，也就是以千为单位使用逗号来分割，使用句点作为小数点。其次，在显示贷款的偿还额之前，改变了数值的显示格式，将精度固定地设置为 2。这使得输出的小数部分有两位，并且正确地取整。如果需要的话，小数部分也可以用 0 来补齐。所有的财务计算程序都使用了相同的方法。为了改变格式以适应不同的语言或者地区，只需要修改传递给 `imbue()` 的 `locale` 对象的语言/地区。

6.2 计算投资的预期价值

另一个常用的财务计算是计算给定初始投资的预期价值、回报率、每年的复利计算数额以及投资的年限。例如，如果本金为 98,000 美元，平均每年返回 6%，您可能想要知道 12 年后您的退休金的数额。这里给出的程序将提供答案。

为了计算这个未来值，使用下面的公式：

$$\text{Future Value} = \text{principal} * ((\text{rateOfRet} / \text{compPerYear}) + 1)^{\text{compPerYear} * \text{numYears}}$$

在此，`rateOfRet` 是返回率，`principal` 包含了原始的投资值，`compPerYear` 指定了每年复利计算期的数目，`numYear` 指定了投资的年限。如果您对 `rateOfRet` 使用按年返回率，那么复利计算期的数目为 1。

在下面的程序中，函数 `futval()` 使用前面的公式来计算投资的未来价值。本金、回报率、投资的年限以及每年复利计算的数量传递给这个函数。函数返回预期价值。

```
#include <iostream>
#include <cmath>
#include <iomanip>
#include <locale>

using namespace std;

// Compute the future value of an investment.
double futval(double principal, double rateOfRet,
              int numYears, int compPerYear) {
    double b, e;

    rateOfRet /= 100.0; // convert percentage to fraction

    b = (1 + rateOfRet/compPerYear);
    e = compPerYear * numYears;

    return principal * pow(b, e);
}
```

```

int main() {
    double p, r;
    int y, cpy;

    // Set locale to English. Adjust as necessary
    // for your language and/or region.
    cout.imbue(locale("english"));

    cout << "Enter principal: ";
    cin >> p;

    cout << "Enter rate of return (as a percentage): ";
    cin >> r;

    cout << "Enter number years: ";
    cin >> y;

    cout << "Enter number of compoundings per year: ";
    cin >> cpy;

    cout << "\nFuture value: " << fixed << setprecision(2)
        << futval(p, r, y, cpy) << endl;

    return 0;
}

```

下面是运行示例：

```

Enter principal: 10000
Enter rate of return (as a percentage): 6
Enter number years: 5
Enter number of compoundings per year: 12

Future value: 13,488.50

```

6.3 计算为了获得预期的价值所需的原始投资

有时候您想要知道为了获得预期的价值所需的原始投资的数额。例如，如果您想要为孩子的大学教育储蓄，并且您知道 5 年内需要 75,000 美元，当利率为 7% 的时候，您需要投资多少钱来达到这个目标？下面开发的这个程序可以回答这个问题。

计算原始投资的公式如下：

$$\text{Initial Investment} = \text{targetValue} / (((\text{rateOfRet} / \text{compPerYear}) + 1)^{\text{compPerYear} * \text{numYears}})$$

在此 `rateOfRet` 指定了返回率，`targetValue` 包含了需要的预期价值，`compPerYear` 指定了每年复利计算期的数目，`numYears` 指定了投资的年限。如果您对 `rateOfRet` 使用年返回率，则复利计算期的数目为 1。

在下面的程序中，函数 `initval()` 使用了前面的公式来计算为了获得预期价值所需的原始投资。目标价值、返回率、投资的年限以及每年复利计算的数目都传递给这个函数。函数返回为了获得预期价值所需的原始投资。

```
#include <iostream>
#include <cmath>
#include <iomanip>
#include <locale>

using namespace std;

// Compute the initial investment necessary for
// a specified future value.
double initval(double targetValue, double rateOfRet,
               int numYears, int compPerYear) {
    double b, e;
    rateOfRet /= 100.0; // convert percentage to fraction

    b = (1 + rateOfRet/compPerYear);
    e = compPerYear * numYears;

    return targetValue / pow(b, e);
}

int main() {
    double p, r;
    int y, cpy;

    // Set locale to English. Adjust as necessary
    // for your language and/or region.
    cout.imbue(locale("english"));

    cout << "Enter desired future value: ";
    cin >> p;

    cout << "Enter rate of return: ";
    cin >> r;

    cout << "Enter number years: ";
    cin >> y;

    cout << "Enter number of compoundings per year: ";
    cin >> cpy;

    cout << "\nInitial investment required: "
        << fixed << setprecision(2)
        << initval(p, r, y, cpy) << endl;

    return 0;
}
```

下面是运行的样本：

```
Enter desired future value: 75000
Enter rate of return (as a percentage): 7
Enter number years: 5
Enter number of compoundings per year: 4

Initial investment required: 53,011.84
```

6.4 为了获得预期的养老金所需的原始投资

另一个常用的财务计算是计算为了能够定期获取预期数量的养老金必须初期投资的数量。例如，您在退休时可能每个月需要 5,000 美元的退休金，并且需要 20 年。问题是为了保证这些养老金，您需要投资多少钱呢？答案可以用下面的公式计算出来：

$$\text{Initial Investment} = ((\text{regWD} * \text{wdPerYear}) / \text{rateOfRet}) * (1 - (1 / (\text{rateOfRet} / \text{wdPerYear} + 1))^{\text{wdPerYear} * \text{numYears}})$$

在此，`rateOfRet` 指定了返回率，`regWD` 包含了需要的定期提取的数量，`wdPerYear` 指定了每年提取的次数，`numYears` 指定了养老金的年限。

在下面的程序中，函数 `annuity()` 计算了为了获得预期的养老金所需的初期投资。所要提取的数量、返回率、养老金的年限以及每年复利计算数目传递给这个函数。函数返回为了获得养老金所需的最少投资。

```
#include <iostream>
#include <cmath>
#include <iomanip>
#include <locale>

using namespace std;

// Compute the initial investment necessary for
// a desired annuity. In other words, it finds
// the initial amount needed to allow the regular
// withdrawals of a desired amount over a period
// of time.
double annuity(double regWD, double rateOfRet,
               int numYears, int numPerYear) {

    double b, e;
    double t1, t2;

    rateOfRet /= 100.0; // convert percentage to fraction

    t1 = (regWD * numPerYear) / rateOfRet;

    b = (1 + rateOfRet/numPerYear);
    e = numPerYear * numYears;
```

```

    t2 = 1 - (1 / pow(b, e));

    return t1 * t2;
}

int main() {
    double wd, r;
    int y, wpy;

    // Set locale to English. Adjust as necessary
    // for your language and/or region.
    cout.imbue(locale("english"));

    cout << "Enter desired withdrawal: ";
    cin >> wd;

    cout << "Enter rate of return (as a percentage): ";
    cin >> r;

    cout << "Enter number years: ";
    cin >> y;

    cout << "Enter number of withdrawals per year: ";
    cin >> wpy;

    cout << "\nInitial investment required: "
        << fixed << setprecision(2)
        << annuity(wd, r, y, wpy) << endl;

    return 0;
}

```

下面是运行样本：

```

Enter desired withdrawal: 5000
Enter rate of return (as a percentage): 6
Enter number years: 20
Enter number of withdrawals per year: 12

Initial investment required: 697,903.86

```

6.5 计算给定投资所能得到的养老金的最大值

另一个养老金计算问题是计算给定的投资在一段时期之后可以得到的养老金的最大值（定期提取）。例如，如果您有 500,000 美元的退休账户，假定返回率为 6%，20 年内您每个月可以提取多少美元呢？计算最大提取金额的公式如下所示：

$$\text{Maximum Withdrawal} = \text{principal} * \left(\left(\frac{\text{rateOfRet}}{\text{wdPerYear}} \right) / \left(-1 + \left(\left(\frac{\text{rateOfRet}}{\text{wdPerYear}} \right) + 1 \right)^{\text{wdPerYear} * \text{numYears}} \right) \right) + \left(\frac{\text{rateOfRet}}{\text{wdPerYear}} \right)$$

其中 `rateOfRet` 指定了返回率, `principal` 包含了原始投资值, `wdPerYear` 指定了每年提取的次数, `numYears` 指定了退休金的年限。

下面所示的程序中, 函数 `maxwd()` 计算了在给定返回率和年限的时候, 可以定期提取的最大值。将本金、返回率、退休金年限以及每年复利计算的数目传递给这个函数。函数返回最大的养老金数额。

```
#include <iostream>
#include <cmath>
#include <iomanip>
#include <locale>

using namespace std;

// Compute the maximum annuity that can
// be withdrawn from an investment over
// a period of time.
double maxwd(double principal, double rateOfRet,
              int numYears, int numPerYear) {

    double b, e;
    double t1, t2;

    rateOfRet /= 100.0; // convert percentage to fraction

    t1 = rateOfRet / numPerYear;

    b = (1 + t1);
    e = numPerYear * numYears;

    t2 = pow(b, e) - 1;

    return principal * (t1/t2 + t1);
}

int main() {
    double p, r;
    int y, wpy;

    // Set locale to English. Adjust as necessary
    // for your language and/or region.
    cout.imbue(locale("english"));

    cout << "Enter principal: ";
    cin >> p;

    cout << "Enter rate of return (as a percentage): ";
```

```

    cin >> r;

    cout << "Enter number years: ";
    cin >> y;

    cout << "Enter number of withdrawals per year: ";
    cin >> wpy;

    cout << "\nMaximum withdrawal: " << fixed << setprecision(2)
        << maxwd(p, r, y, wpy) << endl;

    return 0;
}

```

下面是运行样本：

```

Enter principal: 500000
Enter rate of return (as a percentage): 6
Enter number years: 20
Enter number of withdrawals per year: 12

Maximum withdrawal: 3,582.16

```

6.6 计算贷款余额

您经常想知道贷款的余额。如果您知道本金、利率、贷款期限以及还款的数量，这很容易计算。为了计算贷款余额，您必须对还款数量求和，减去每次还款所分配的利息，然后从本金中减去这个结果。

这里所给出的 `balance()` 函数实现贷款计算的余额。将本金、利率、还款数量、每年偿还的次数以及已经偿还的次数传递给这个函数。函数返回贷款余额。

```

#include <iostream>
#include <cmath>
#include <iomanip>
#include <locale>

using namespace std;

// Find the remaining balance on a loan.
double balance(double principal, double intRate,
               double payment, int payPerYear,
               int numPayments) {

    double bal = principal;
    double rate = intRate / payPerYear;

    rate /= 100.0; // convert percentage to fraction

    for(int i = 0; i < numPayments; i++)
        bal -= payment - (bal * rate);
}

```

```

    return bal;
}

int main() {
    double p, r, pmt;
    int ppy, npmt;

    // Set locale to English. Adjust as necessary
    // for your language and/or region.
    cout.imbue(locale("english"));

    cout << "Enter original principal: ";
    cin >> p;

    cout << "Enter interest rate (as a percentage): ";
    cin >> r;

    cout << "Enter payment: ";
    cin >> pmt;

    cout << "Enter number of payments per year: ";
    cin >> ppy;

    cout << "Enter number of payments made: ";
    cin >> npmt;

    cout << "Remaining balance: " << fixed << setprecision(2)
        << balance(p, r, pmt, ppy, npmt) << endl;

    return 0;
}

```

下面是运行样本：

```

Enter original principal: 10000
Enter interest rate (as a percentage): 9
Enter payment: 207.58
Enter number of payments per year: 12
Enter number of payments made: 30

Remaining balance: 5,558.19

```

6.7 尝试完成以下任务

您可能还会发现许多有用的财务计算。例如，计算投资的返回率或者计算为了达到某个未来价值所需的定期存款都是有用的程序。您还可打印一个贷款分期偿还的图表。

您可能会尝试建立一个比较大的应用程序来包含本章提供的所有计算，使得用户从菜单中选择需要的计算。

第 7 章 基于 AI 的问题求解

本章讲述一个有趣的程序设计主题：人工智能(Artificial Intelligence, AI)。本书的目的之一是显示 C++ 的适用范围以及它所具有的多种功能。或许一个需要人工智能领域的应用程序最能说明这个问题。

人工智能的领域由几个引人注目的部分组成，但是许多基于 AI 的应用程序的基础是问题求解。本质上，存在两类问题。第一种类型的问题可以用某种确定性的、一定能够成功的程序来解决，如计算某个角度的正弦或者某个值的平方根。这种类型的问题很容易转换为计算机可以处理的算术。然而，实际上很少有问题会这么直接。相反，许多类型的问题只有通过寻找一个解决方案来解决。AI 所关心的正是这类问题的解决方案。这也是本章讲述的搜索类型。

为了理解搜索对 AI 如此重要的原因，请考虑下面的问题。AI 搜索早期的目的之一是创建一个通用问题求解程序。通用问题求解程序是一个程序，这个程序可以给出各种不同问题的解决方案，而不需要有特定的、设计好的知识。可以理解，这种程序非常令人满意。遗憾的是，这种通用问题求解程序非常难于实现，它是可望而不可及的。一个麻烦是许多实际问题具有不同的尺寸和复杂度。因为通用问题求解程序必须在一个非常大的、具有复杂可能性的世界中搜索一个解决方案，所以首先必须找到某种在这种环境下搜索的优先方法。在本章我们并没有雄心勃勃地试图开发一个通用问题求解程序，我们将探索基于 AI 的搜索技术，这种技术可以应用于许多问题。

C++ 是非常适合于 AI 开发人员的语言。原因是 C++ 提供了对基于 AI 的应用程序通常使用的程序元素的支持：递归、链表和堆栈。正如您所知道的那样，C++ 可以方便而高效地处理递归。加上各种 STL 容器所提供的功能，您具有一个有效的 AI 开发环境。

7.1 表示法和术语

假定您丢失了汽车钥匙。您知道它在房间的某个位置，如图 7-1 所示。

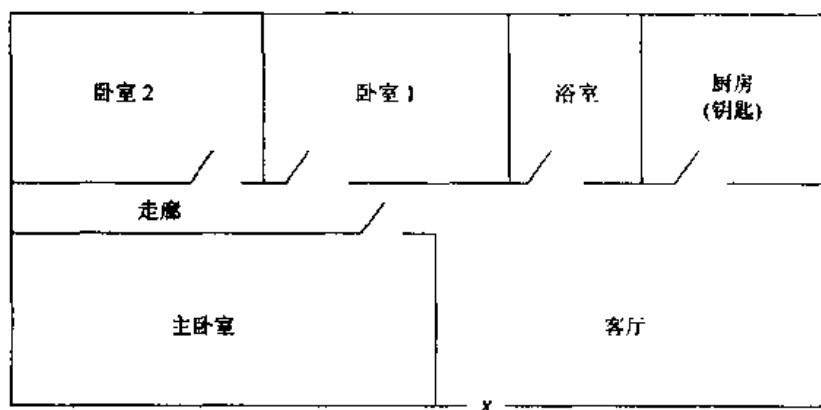


图 7-1 钥匙的位置

您正站在前门(X 所指的位置), 当您开始查找时, 先检查了客厅。然后走过走廊, 到第一间卧室, 通过走廊到第二间卧室, 再返回走廊, 然后去主卧室。仍然没有找到钥匙, 您又返回到客厅。最后, 您发现钥匙在厨房。这种情况很容易用图表来表示, 如图 7-2 用图表的形式来表示这个搜索问题是有帮助的, 因为它提供了一种方便的方法来描述找到解决方案的方式。

记住前面的讨论, 考虑表 7-1 所示的术语, 这些术语将在本章使用:

表 7-1 搜索算法的相关术语

术 语	说 明
节点	一个不连续的点
终端节点	结束路径的节点
搜索空间	所有节点的集合
目标	代表搜索目标的节点
试探法	能够决定某个节点是否比另一个节点更好的信息
解决路径	通向目标的路径上的节点的有向图

在丢失钥匙的示例中, 住宅中的每个房间都是一个节点; 整个住宅是搜索空间; 目标是厨房; 解决路径如图 7-2 所示。卧室、厨房以及浴室都是终端节点, 因为它们不通向任何地方。图中没有表示试探法, 它们是您可能用来更好地选择路径的技术。

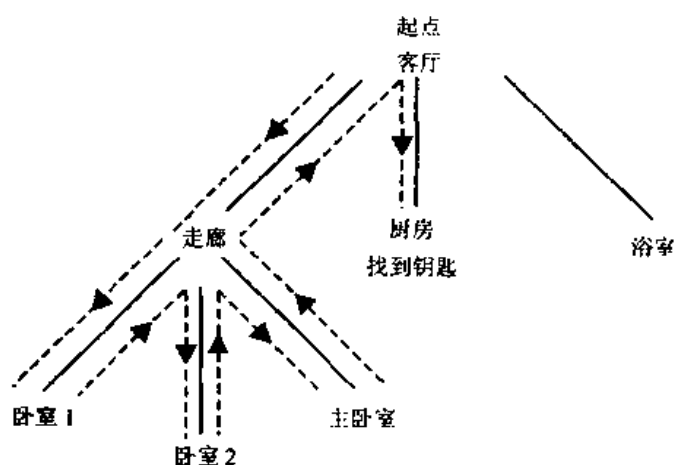


图 7-2 寻找丢失钥匙的解决路径

7.2 组合爆炸

通过前面的示例, 您可能会认为解决方案的查找很容易——从起点开始, 逐一搜索获取结果。在特别简单的丢失钥匙的示例中, 这是一种不错的方法, 因为搜索空间非常小。但是对于许多问题(特别是您需要使用计算机解决的那些问题), 搜索空间中节点的数目非常多, 随着搜索空间的增加, 通向目标的可能路径也在增加。问题在于, 向搜索空间中加入一个节点时, 往往加入了不止一条路径。也就是说, 潜在的通向目标的路径数量随着搜索空间尺寸的增长以非线性的方式在增长。在非线性的情况下, 可能的路径数量会迅速地变得很大。

例如，考虑在表格中排列下面 3 个对象——A、B 和 C——的方法的数量。有 6 种可能的排列方式如下所示。

A	B	C
A	C	B
B	C	A
B	A	C
C	B	A
C	A	B

可以迅速证明，这 6 种方式是 A、B 和 C 全部的排列方式。您可以使用组合数学——研究事物组合方法的学科——的定理得到相同的结果。根据这个定理，排列 N 个对象的方式的数量为 $N!$ (N 的阶乘)。数字的阶乘等于这个数字和小于这个数字直到 1 的所有数字的乘积。即 $3!$ 是 $3 \times 2 \times 1$ ，等于 6。如果您排列 4 个对象，那么排列方式的数量就是 $4!$ ，即 24。如果有 5 个对象，那么这个数量为 120，6 个对象的数量为 720。1000 个对象的可能的排列方式的数量是巨大的。图 7-3 中的图表给出了被称为组合爆炸的直观的概念。一旦可能性多了起来，验证(甚至是穷举)所有的排列迅速地变得困难起来。

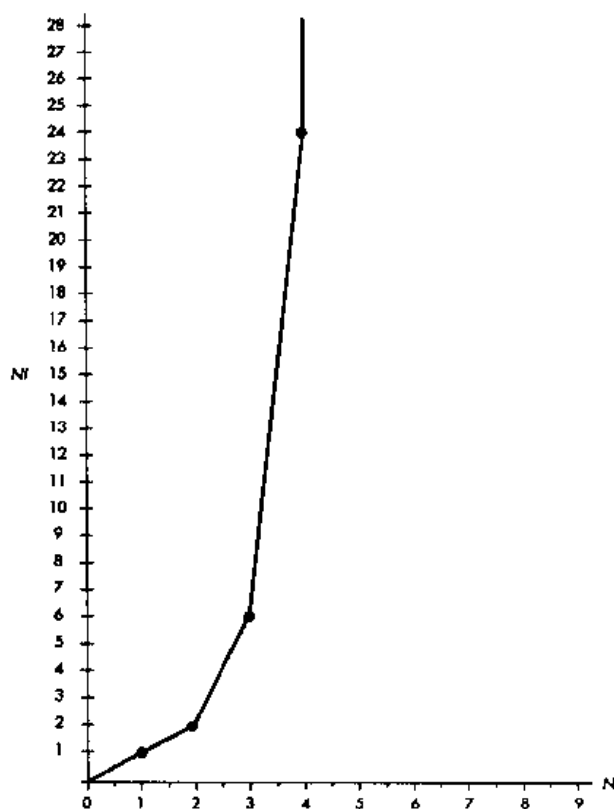


图 7-3 关于阶乘的组合激增

搜索空间的路径也存在类似的组合爆炸。因此，只有非常简单的问题可以用穷举搜索。穷举搜索是检查所有节点的搜索方法。因此，这是一种“暴力”的技术。暴力总是有效的，但是对于比较大的问题，它往往是不实际的，因为它消耗太多的时间，或者太多的计算机资源，或者二者兼有。为此提出了基于 AI 的搜索技术。

7.3 搜索方法

有多种方式搜索一个解决方案。最基本的 4 种方式为：

- 深度优先
- 广度优先
- 爬山法
- 最低成本法

本章讲述了每种搜索方法。

搜索方法性能评估

评估一个基于 AI 的搜索方法的性能非常复杂。幸运的是，对于本章的目的而言，我们只需要关心下面两个尺度：

- 发现解决方案的速度
- 解决方案的质量

有这样一些问题，对这些问题而言最关键的是用最小的努力找到一个解决方案，任何方案都可以。对于这种问题，第一个尺度尤其重要。在其他情况下，解决方案的质量更加重要。

搜索的速度受搜索空间的大小以及在寻找解决方案的过程中实际经过的节点数目的影响。由于从死胡同返回是一种不必要的消耗，因此需要一种很少折回的搜索方法。

在基于 AI 的搜索中，寻找最优解决方案与寻找好的解决方案不同。寻找最优的解决方案需要使用穷举搜索，因为有时这是确定已经找到最优的解决方案的惟一方法。相反，寻找一个好的解决方案意味着在一套约束内寻找解决方案——是否存在更好的解决方案并不重要。

正如您所看到的那样，本章所描述的搜索技术在某种情况下比在其他情况下好。很难说某种搜索方法总是比其他的方法优越，但是对于平均情况，有些搜索技术很可能会更好。另外，问题的定义方式有时也会有助于选择合适的搜索方法。

7.4 需要解决的问题

现在，考虑我们将使用不同的搜索方法来解决的问题。假定您是一位旅行代理人，一个相当挑剔的客户想让您预定 XYZ 航空公司的从纽约到洛杉矶的航班。您告诉客户 XYZ 没有直接从纽约到洛杉矶的航班，但是客户坚持说，XYZ 是他惟一想要乘坐的飞机的航空公司。因此，您必须寻找连接纽约和洛杉矶之间的航线。您可以参考 XYZ 的定期航班，如表 7-2 所示。

表 7-2 航班和距离

航 班	距 离
纽约到芝加哥	900 英里
芝加哥到丹佛	1000 英里
纽约到多伦多	500 英里
纽约到丹佛	1800 英里
多伦多到卡尔加里	1700 英里

(续表)

航 班	距 离
多伦多到洛杉矶	2500 英里
多伦多到芝加哥	500 英里
丹佛到乌尔班纳	1000 英里
丹佛到休斯顿	1000 英里
休斯顿到洛杉矶	1500 英里
丹佛到洛杉矶	1000 英里

您很快看到可以使您的客户从纽约飞到洛杉矶的连接。问题在于如何把您脑海里想到的内容编写为 C++ 程序。

图的表示

XYZ 时间表上的航班信息可以转换为有向图, 如图 7-4 所示。有向图是一幅简单的图形, 其中使用线段来连接每个节点, 并使用箭头指示动作的方向。在有向图中, 不能逆着箭头的方向旅行。



图 7-4 XYZ 的定期航班的有向图

为了使得问题易于理解, 在图 7-5 中以树形格式重新绘制了这幅图。本章的剩余部分使用图 7-5。目标(洛杉矶)被圈了起来。另外还要注意, 为了简化图的结构, 各个城市都出现了不止一次。因此, 树形的表示不是表述了一个二叉树, 而只是为了便于分析。

现在我们准备开发不同的搜索技术来查找从纽约到洛杉矶的航线。

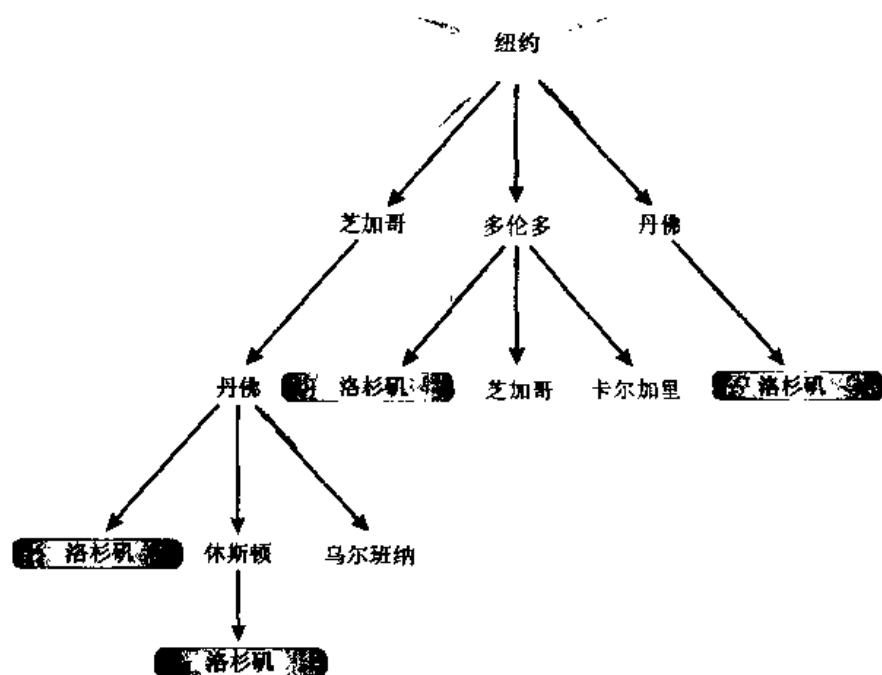


图 7-5 XYZ 定期航班的树形图

7.5 FlightInfo 结构和 Search 类

编写一个寻找从纽约到洛杉矶的航线的程序需要一个包含了航班信息的数据库。数据库中的每一个记录项都必须包含出发的城市和目标城市、两个城市之间的距离，以及帮助沿原路返回的标记。这些信息保存在名为 FlightInfo 的结构中，如下所示：

```

// Flight information.
struct FlightInfo {
    string from; // departure city
    string to; // destination city
    int distance; // distance between from and to
    bool skip; // used in backtracking

    FlightInfo() {
        from = "";
        to = "";
        distance = 0;
        skip = false;
    }

    FlightInfo(string f, string t, int d) {
        from = f;
        to = t;
        distance = d;
        skip = false;
    }

```

```
}  
};
```

本章剩余部分所描述的全部搜索方法都用到这个结构。

基于 AI 的搜索被封装到名为 **Search** 的类中。这个类精确的实现随搜索方法的不同而不同。然而，所有的版本都具有相同的常规架构，如下所示：

```
// An AI-based search class.  
class Search {  
    // This vector holds the flight information.  
    vector<FlightInfo> flights;  
  
    // This stack is used for backtracking.  
    stack<FlightInfo> btStack;  
  
    // If there is a flight between from and to,  
    // store the distance of the flight in dist.  
    // Return true if the flight exists and,  
    // false otherwise.  
    bool match(string from, string to, int &dist);  
  
    // Given from, find any connection.  
    // Return true if a connection is found,  
    // and false otherwise.  
    bool find(string from, FlightInfo &f);  
  
public:  
  
    // Put flights into the database.  
    void addflight(string from, string to, int dist) {  
        flights.push_back(FlightInfo(from, to, dist));  
    }  
  
    // Show the route and total distance.  
    void route();  
  
    // Determine if there is a route between from and to.  
    void findroute(string from, string to);  
  
    // Return true if a route has been found.  
    bool routefound() {  
        return !btStack.empty();  
    }  
};
```

Search 声明了两个私有实例变量。第一个是 **FlightInfo** 对象的 **vector** 对象，名为 **flights**，保存航班的数据。（**vector** 是一个实现动态数组的 STL 容器）。第二个是一个堆栈，名为 **btStack**，用它来沿原路返回。您将会看到，对于所有的搜索方法，这个堆栈都很重要。

Search 包含了两个内联函数：**addflight()**和 **routefound()**函数。当第一次创建 **Search** 对象时，它的 **flights** 向量是空的。连接是通过重复地调用 **addflight()**，加入指定出发城市、目标城市以

及两个城市之间的距离。addflight()函数简单地把每个连接放入 flights 向量的结尾, 这个向量的大小会自动扩展以适应新的条目。

routefound()函数基于返回堆栈中的内容判断出发城市和目标城市之间的航线是否存在。如果堆栈返回为空, 两个城市之间不存在航线。否则, 返回堆栈包含这个航线(创建航线的过程随搜索方法的不同而不同)。

Search 其余成员的实现在下节描述。表 7-3 简要描述了这些成员函数的功能。

表 7-3 Search 其余成员的目的

Search 其余成员	目 的
match()	判断两个城市之间是否有直接的连接
find()	尝试寻找指定城市到任何其他城市的连接
findroute()	尝试在出发城市和目标城市之间找到一条航线
route()	显示航线

7.6 深度优先搜索

深度优先搜索在尝试另一种路径之前, 会探索每条可能的路径, 直到结束。为了理解它确切的运行方式, 考虑图 7-6 所示的树形结构。F 是目标。

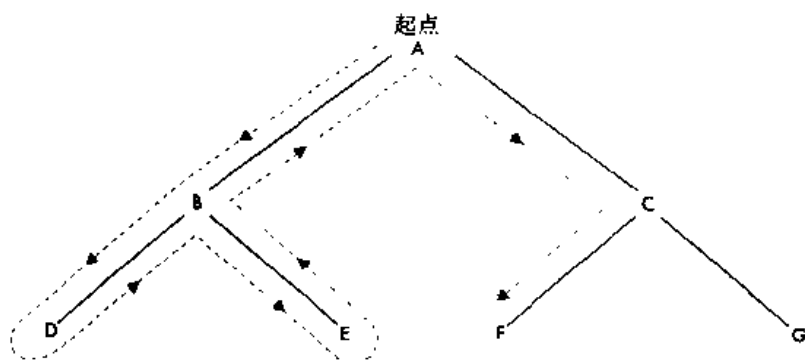


图 7-6 树形结构

深度优先搜索以下面的顺序遍历图 7-6: ABDBEBACF。如果您对树熟悉的话, 应该知道这种搜索方法是树的中序遍历。也就是说, 路径一直向左, 直到遇到终端节点或者找到目标。如果到达终端节点, 路径回退到上一层, 转向右边, 然后再向左边搜索, 直到遇到目标或者终端节点。这个过程一直持续到发现目标或者检查到搜索空间中的最后一个节点。

正如您所看到的那样, 深度优先搜索总是能够找到目标, 因为在最坏的情况下, 它退化为穷举搜索。在这个示例中, 如果 G 是目标, 那么就会导致穷举搜索。

整个深度优先搜索程序如下所示:

```
// Search for a route.
#include <iostream>
#include <stack>
#include <string>
#include <vector>
```



```

using namespace std;

// Flight information.
struct FlightInfo {
    string from; // departure city
    string to; // destination city
    int distance; // distance between from and to
    bool skip; // used in backtracking

    FlightInfo() {
        from = "";
        to = "";
        distance = 0;
        skip = false;
    }

    FlightInfo(string f, string t, int d) {
        from = f;
        to = t;
        distance = d;
        skip = false;
    }
};

// Find connections using a depth-first search.
class Search {
    // This vector holds the flight information.
    vector<FlightInfo> flights;

    // This stack is used for backtracking.
    stack<FlightInfo> btStack;

    // If there is a flight between from and to,
    // store the distance of the flight in dist.
    // Return true if the flight exists and
    // false otherwise.
    bool match(string from, string to, int &dist);

    // Given from, find any connection.
    // Return true if a connection is found,
    // and false otherwise.
    bool find(string from, FlightInfo &f);

public:
    // Put flights into the database.
    void addflight(string from, string to, int dist) {
        flights.push_back(FlightInfo(from, to, dist));
    }
};

```

```

// Show the route and total distance.
void route();

// Determine if there is a route between from and to.
void findroute(string from, string to);

// Return true if a route has been found.
bool routefound() {
    return !btStack.empty();
}
};

// Show the route and total distance.
void Search::route()
{
    stack<FlightInfo> rev;
    int dist = 0;
    FlightInfo f;

    // Reverse the stack to display route.
    while(!btStack.empty()) {
        f = btStack.top();
        rev.push(f);
        btStack.pop();
    }

    // Display the route.
    while(!rev.empty()) {
        f = rev.top();
        rev.pop();
        cout << f.from << " to ";
        dist += f.distance;
    }

    cout << f.to << endl;
    cout << "Distance is " << dist << endl;
}

// If there is a flight between from and to,
// store the distance of the flight in dist.
// Return true if the flight exists and,
// false otherwise.
bool Search::match(string from, string to, int &dist)
{
    for(unsigned i=0; i < flights.size(); i++) {
        if(flights[i].from == from &&
            flights[i].to == to && !flights[i].skip)
        {
            flights[i].skip = true; // prevent reuse

```

```

        dist = flights[i].distance;
        return true;
    }
}

return false; // not found
}

// Given from, find any connection.
// Return true if a connection is found,
// and false otherwise.
bool Search::find(string from, FlightInfo &f)
{
    for(unsigned i=0; i < flights.size(); i++) {
        if(flights[i].from == from && !flights[i].skip) {
            f = flights[i];
            flights[i].skip = true; // prevent reuse

            return true;
        }
    }

    return false;
}

// Depth-first version.
// Determine if there is a route between from and to.
void Search::findroute(string from, string to)
{
    int dist;
    FlightInfo f;

    // See if at destination.
    if(match(from, to, dist)) {
        btStack.push(FlightInfo(from, to, dist));
        return;
    }

    // Try another connection.
    if(find(from, f)) {
        btStack.push(FlightInfo(from, to, f.distance));
        findroute(f.to, to);
    }
    else if(!btStack.empty()) {
        // Backtrack and try another connection.
        f = btStack.top();
        btStack.pop();
        findroute(f.from, f.to);
    }
}

```

```

int main() {
    char to[40], from[40];
    Search ob;

    // Add flight connections to database.
    ob.addflight("New York", "Chicago", 900);
    ob.addflight("Chicago", "Denver", 1000);
    ob.addflight("New York", "Toronto", 500);
    ob.addflight("New York", "Denver", 1800);
    ob.addflight("Toronto", "Calgary", 1700);
    ob.addflight("Toronto", "Los Angeles", 2500);
    ob.addflight("Toronto", "Chicago", 500);
    ob.addflight("Denver", "Urbana", 1000);
    ob.addflight("Denver", "Houston", 1000);
    ob.addflight("Houston", "Los Angeles", 1500);
    ob.addflight("Denver", "Los Angeles", 1000);

    // Get departure and destination cities.
    cout << "From? ";

    cin.getline(from, 40);
    cout << "To? ";

    cin.getline(to, 40);

    // See if there is a route between from and to.
    ob.findroute(from, to);

    // If there is a route, show it.
    if(ob.routefound())
        ob.route();

    return 0;
}

```

注意 `main()` 提示您输入出发城市和目标城市。这意味着可以使用这个程序来寻找任何两个城市之间的航线。然而，本章的其余部分假定纽约是出发城市，洛杉矶是目标城市。

当程序以纽约作为出发城市、洛杉矶作为目标城市运行的时候，会显示如下的解决方案：

```

From? New York
To? Los Angeles
New York to Chicago to Denver to Los Angeles
Distance is 2900

```

如果您参考图 7-7，就会发现这实际上是深度优先搜索发现的第一个解决方案。这也是相当不错的一个解决方案。

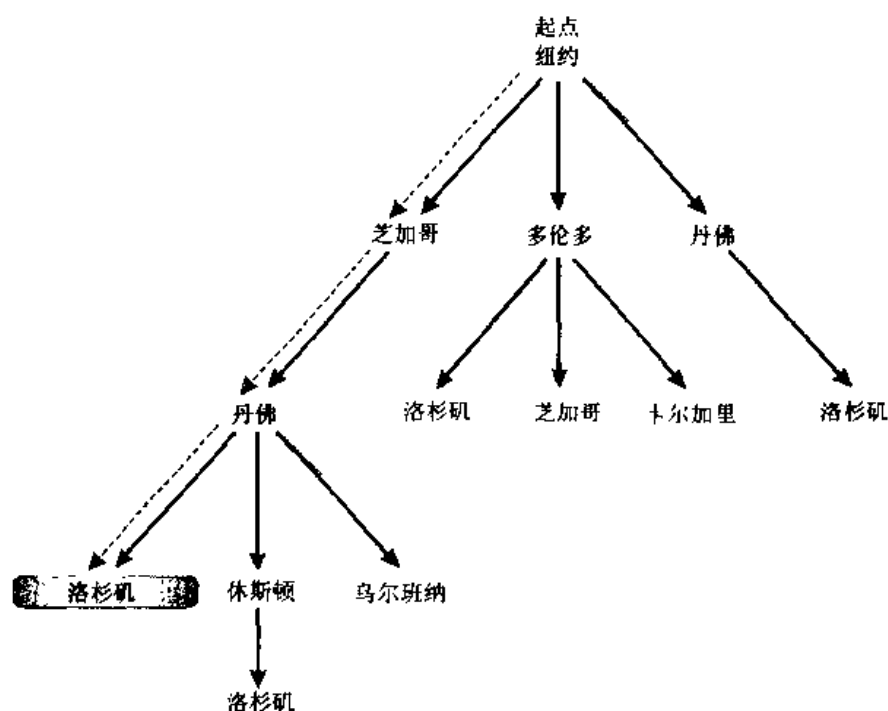


图 7-7 寻找解决方案的深度优先路径

正如 `main()` 函数显示的那样，为了使用 `Search`，首先创建了一个 `Search` 对象。然后用这个连接加载航班数据库。随后，调用 `findroute()` 试图在出发城市与目标城市之间寻找一条航线。为了判断是否找到了这样的航线，调用了 `routeFound()` 函数。如果存在这样的航线，这个函数返回 `true`。为了显示这条航线，调用了 `route()` 函数。现在让我们仔细分析深度优先搜索的每个部分。

7.6.1 `match()` 函数

这里所给出的 `match()` 函数实现判断两个城市之间是否有航班，这两个城市分别由 `from` 和 `to` 指定：

```

// If there is a flight between from and to,
// store the distance of the flight in dist.
// Return true if the flight exists and
// false otherwise.
bool Search::match(string from, string to, int &dist)
{
    for(unsigned i=0; i < flights.size(); i++) {
        if(flights[i].from == from &&
           flights[i].to == to && !flights[i].skip)
        {
            flights[i].skip = true; // prevent reuse
            dist = flights[i].distance;
            return true;
        }
    }

    return false; // not found
}

```

`match()`函数通过搜索 `flights` 向量来操作，在其中查找是否存在与 `from` 和 `to` 匹配的出发城市与目标城市的条目。如果没有这样的航班，则返回 `false`。如果存在这样的航班，则返回 `true`，在此情况下还会获取两个城市之间的距离，并将这个值存储在 `dist` 参数所引用的变量中。

注意 `match()`忽略了 `skip` 字段为 `true` 的连接。另外，如果找到了某个连接，就会设置 `skip` 字段。用它来管理从死胡同的返回，从而避免一次又一次地测试一个相同的连接。

7.6.2 `find()`函数

`find()`函数如下所示，用来在数据库中搜索从给定的出发城市开始的连接：

```
// Given from, find any connection.
// Return true if a connection is found,
// and false otherwise.
bool Search::find(string from, FlightInfo &f)
{
    for(unsigned i=0; i < flights.size(); i++) {
        if(flights[i].from == from && !flights[i].skip) {
            f = flights[i];
            flights[i].skip = true; // prevent reuse
            return true;
        }
    }

    return false;
}
```

出发城市传递给 `from`。如果找到了一个连接航班，则 `find()`返回 `true`，与这个连接相关的 `FlightInfo` 对象存储在第二个参数 `f` 引用的变量中。否则，`find()`返回 `false`。`match()`与 `find()`之间的区别是 `match()`判断指定的两个城市之间是否存在航班；`find()`判断是否存在从给定城市到其他任何城市的航班。

如同 `match()`一样，`find()`也使用 `skip` 字段来管理从死胡同的返回。

7.6.3 `findroute()`函数

实际寻找连接航班的代码包含在 `findroute()`函数中，它是寻找两个城市之间航线的关键例程。函数调用时需要出发城市和目标城市的名称。

```
// Determine if there is a route between from and to
// by using depth-first searching.
void Search::findroute(string from, string to)
{
    int dist;
    FlightInfo f;

    // See if at destination.
    if(match(from, to, dist)) {
        btStack.push(FlightInfo(from, to, dist));
        return;
    }
}
```

```

// Try another connection.
if(find(from, f)) {
    btStack.push(FlightInfo(from, to, f.distance));
    findroute(f.to, to);
}
else if(!btStack.empty()) {
    // Backtrack and try another connection.
    f = btStack.top();
    btStack.pop();
    findroute(f.from, f.to);
}
}
}

```

让我们仔细分析 `findroute()`。首先，通过 `match()` 检查航班数据库，以判断是否有从 `from` 到 `to` 的航班。如果有，那么已经到达目标，这个连接被压入堆栈，函数返回。否则，`findroute()` 调用 `find()` 来查找 `from` 与其他任何城市之间的连接。如果找到这样的连接，则 `find()` 函数返回 `true`，在此情况下，存储 `f` 中描述这个连接的 `FlightInfo` 对象。如果没有有效的连接航班，则 `find()` 返回 `false`。如果存在连接航班，当前的航班被压入返回堆栈中，并递归调用 `findroute()`，`f.to` 中的城市变为新的出发城市。否则，发生沿原路返回。前面的节点从堆栈中删除，并递归调用 `findroute()`。这个过程一直持续到发现目标，或者穷举尽数据库。

例如，如果使用纽约和芝加哥作为参数来调用 `findroute()`，第一个 `if` 将成功，`findroute()` 将终止，因为纽约到芝加哥有直接的航班。当使用纽约和卡尔加里作为参数来调用 `findroute()` 时，情况变得复杂起来。在此情况下，第一个 `if` 将会失败，因为这两个城市之间没有直接的航班。随后，尝试寻找纽约与任何其他城市之间的连接来测试第二个 `if`。在此情况下，`find()` 首先找到纽约到芝加哥的连接，这个连接压入到返回堆栈中，然后使用芝加哥作为起点来递归调用 `findroute()`。遗憾的是，在此没有从芝加哥到卡尔加里的路线，随后是几条错误的路线。最终，在几次递归调用 `findroute()` 以及沿原路返回之后，发现了从纽约到多伦多的连接，以及多伦多与卡尔加里的连接。从而返回 `findroute()`，并将这个过程中所有的递归调用解开。最后，返回对 `findroute()` 的原始调用。您或许会想要在 `findroute()` 中加入 `cout` 语句，来准确地观察使用其他不同的出发城市和目标城市时，这个函数的运行方式。

`findroute()` 实际上并没有返回解决方案——它生成了解决方案，理解这一点很重要。一旦从 `findroute()` 退出，返回堆栈中就包含了从出发点到目标点的路线。也就是说，解决方案保存在 `btStack` 中。另外，`findroute()` 的成功或者失败由堆栈的状态决定。空的堆栈意味着失败，否则，堆栈会保存一个解决方案。

通常，返回是基于 AI 的搜索技术中的很重要因素。在 `Search` 中，返回是通过使用递归和返回堆栈来完成的(这就是 C++ 对递归和 STL 的支持使得它成为 AI 开发一个好的选择的原因)。几乎所有的返回的情况都类似于堆栈操作——也就是先进后出。当探索路径时，遇到节点就将其压入堆栈。在每个死胡同，最后的节点弹出堆栈，从这个点开始尝试新的路径。这个过程一直持续，直到到达目标或者搜索完所有的路径。

7.6.4 显示路线

route()函数显示路径和总的距离。这个函数如下所示:

```
// Show the route and total distance.
void Search::route()
{
    stack<FlightInfo> rev;
    int dist = 0;
    FlightInfo f;

    // Reverse the stack to display route.
    while(!btStack.empty()) {
        f = btStack.top();
        rev.push(f);
        btStack.pop();
    }

    // Display the route.
    while(!rev.empty()) {
        f = rev.top();
        rev.pop();
        cout << f.from << " to ";
        dist += f.distance;
    }

    cout << f.to << endl;
    cout << "Distance is " << dist << endl;
}
```

注意第二个名为 rev 的堆栈的使用。在 btStack 中存储的解决方案顺序相反, 这个堆栈的栈顶保存着最后的连接, 栈底保存着第一个连接。为此, 必须将其反转, 从而能够以正确的顺序来显示这个连接。为了将这个解决方案的顺序调整正确, 必须将这个连接从 btStack 中弹出, 并压入 rev 中。

7.6.5 深度优先搜索分析

深度优先搜索找到了一个好的解决方案。另外, 对于这个特定的问题, 深度优先搜索在第一次尝试的时候就找到了这个解决方案, 而没有返回——这很好。然而, 如果这些数据以不同的方式组织, 寻找一个解决方案可能会涉及到相当多的返回。因此, 这个示例的结果不能推广。更重要的是, 当探索一个在末端没有解决方案的大的分支时, 深度优先搜索的性能相当差。在此情况下, 深度优先搜索不仅在探索这个分支上浪费了时间, 在向目标返回时也是如此。

7.7 广度优先搜索

与深度优先搜索相对的是广度优先搜索。在这种方法中, 在搜索下一层的节点之前, 会检查这个层的每个节点。这种遍历方法如图 7-8 所示, C 为目标。

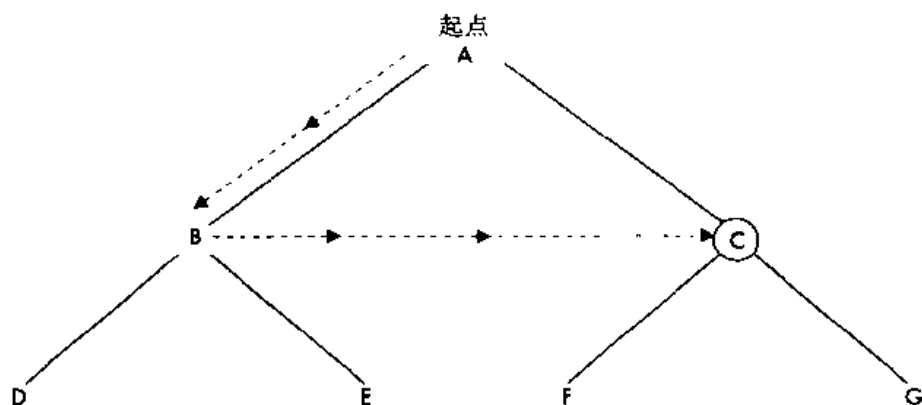


图 7-8 广度优先搜索的遍历方法

尽管二叉树结构的搜索空间很容易描述广度优先搜索的动作，然而许多的搜索空间，包括航班示例，并不是二叉树。因此，“广度”确切的组成有一点主观，是由手头的问题决定的。当涉及到航班示例时，广度优先搜索是通过检查是否存在离开出发城市的航班与到达目标城市的航班之间的连接来实现的。换句话说，在前进到另一层之前，会检查连接航班的所有连接的目的地。

为了使得 Search 执行广度优先搜索，需要修改 findroute() 函数如下所示：

```
// Breadth-first version.
// Determine if there is a route between from and to.
void Search::findroute(string from, string to)
{
    int dist;
    FlightInfo f;

    // This stack is needed by the breadth-first search.
    stack<FlightInfo> resetStck;

    // See if at destination.
    if(match(from, to, dist)) {
        btStck.push(FlightInfo(from, to, dist));
        return;
    }

    // Following is the first part of the breadth-first
    // modification. It checks all connecting flights
    // from a specified node.
    while(find(from, f)) {
        resetStck.push(f);
        if(match(f.to, to, dist)) {
            resetStck.push(FlightInfo(f));
            btStck.push(FlightInfo(from, f.to, f.distance));
            btStck.push(FlightInfo(f.to, to, dist));
            return;
        }
    }
}

// The following code resets the skip fields set by
```

```

// preceding while loop. This is also part of the
// breadth-first modification.
while(!resetStck.empty()) {
    resetSkip(resetStck.top());
    resetStck.pop();
}

// Try another connection.
if(find(from, f)) {
    btStack.push(FlightInfo(from, to, f.distance));
    findroute(f.to, to);
}
else if(!btStack.empty()) {
    // Backtrack and try another connection.
    f = btStack.top();
    btStack.pop();
    findroute(f.from, f.to);
}
}

```

在此做了两个修改。首先，for 循环检查离开出发城市的所有航班，以判断它们是否与到达目标城市的航班相连接。随后，如果没有找到目标，通过调用 `resetSkip()`，将这些连接航班的 `skip` 字段清除，`resetSkip()` 是必须加入到 `Search` 的一个新函数。需要重新设置的连接存储在它们自己的名为 `resetStck` 的堆栈中，这是 `findroute()` 的局部变量。为了能够使用可能涉及到这些连接的其他路径，必须重新设置 `skip` 标志。

`resetSkip()` 函数如下所示：

```

// Reset the skip fields in flights vector.
void Search::resetSkip(FlightInfo f) {
    for(unsigned i=0; i < flights.size(); i++)
        if(flights[i].from == f.from &&
            flights[i].to == f.to)
            flights[i].skip = false;
}

```

您需要在 `Search` 中加入这个函数(及其原型)。

为了尝试广度优先搜索，在前面的搜索程序中用新版本的 `findroute()` 来替换，并加入 `resetSkip()` 函数。当运行的时候，产生了如下的解决方案：

```

From? New York
To? Los Angeles
New York to Toronto to Los Angeles
Distance is 3000

```

图 7-9 显示了这个解决方案的广度优先路径。

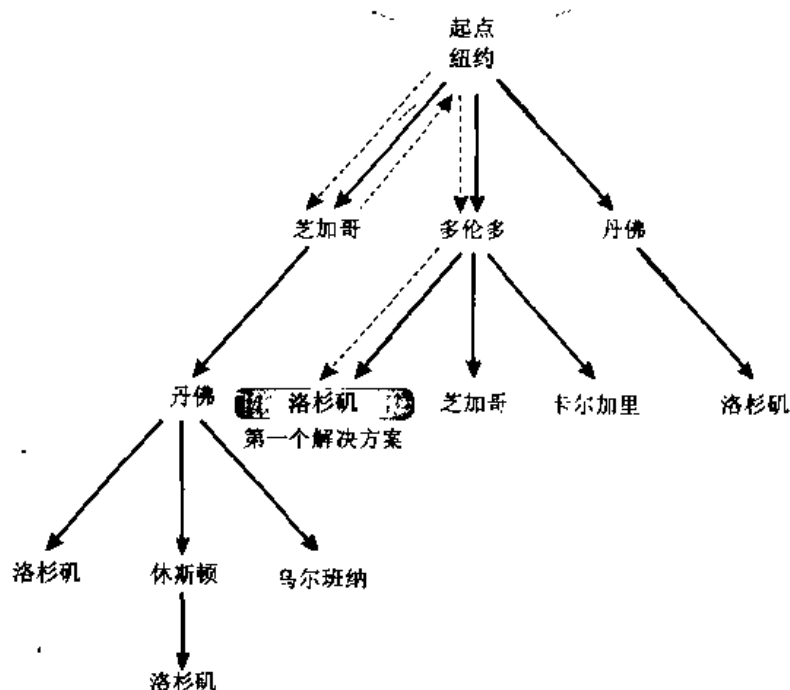


图 7-9 广度优先路径的解决方案

广度优先搜索分析

在这个示例中，广度优先搜索运行的相当好，并找到了一个合理的解决方案。与前面一样，这个结果不能被推广，因为第一条被发现的路径依赖于信息的组织方式。在相同的搜索空间内，深度优先搜索和广度优先搜索经常发现不同的路径，然而这个示例没有显示出这一点。

当目标在搜索空间埋藏地不是很深时，广度优先搜索运行良好。当目标有好几层深的时候，其运行情况比较差。在此情况下，广度优先搜索在返回阶段花费了许多功夫。

7.8 添加启发信息

深度优先搜索和广度优先搜索都没有对搜索空间中的某个节点是否比其他节点更接近于目标做出任何有根据的猜测。相反，它们只是使用先前的模式从一个节点移动到另一个节点，直到最后发现目标。在某些情况下，或许您只能做这么多，但是搜索空间经常包含了一些信息，可以使用这些信息来增加某个搜索比较快速地接近目标的可能性。为了利用这种信息，可以向搜索中增加启发的能力。

启发是用来增加某个搜索朝正确方向前进的可能性的规则。例如，假设您在森林中迷了路，并且需要喝水。森林非常茂密，以至于您不能远眺，这些树木很粗，您不能爬上去观望四周。然而您知道：河流、小溪以及池塘很可能在山谷中；动物们经常制造到水源的路径。当您接近水源时，可能会“闻到”它；您可以听见水流的声音。因此，您首先下山，因为水不大可能在山上。然后您偶然发现鹿的足迹也通向山下。您知道这些足迹可能通向水源，您就沿着这些足迹走。您开始听到细微的流动的声音出现在左边。您知道这可能是水，因此继续朝这个方向移

动。当您移动时，您开始发现空气变得潮湿；您可以闻到水了。最后，您找到了一条小溪，并且喝到了水。在这种情况下，用来找水的启发信息不一定会成功，但是它的确增加了提前成功的可能性。通常，启发信息增加了快速找到目标的可能性。

通常，启发式搜索方法是基于最小化或者最大化某个限制的信息。在安排从纽约到洛杉矶航班的示例中，旅客可能想要最小化两个限制。首先是必须创建的连接的数量。其次是航线的长度。记住，最短的航线不一定意味着最少的连接，反之亦然。在这节中，开发了两个启发式搜索。第一个最小化了连接的数量，第二个最小化了航线的距离。两种启发搜索都创建在深度优先搜索的架构上。

7.8.1 爬山搜索法

试图寻找最小化连接数量的航线的搜索算法使用了这样的启发信息：航班越长，将旅客带往距目标比较近的地方的可能性就越大；因此，连接的数目就会被最小化。用 AI 的语言来说，这是一个爬山的示例。

爬山算法选择距离目标比较近的节点作为下一步(也就是说，距离当前位置最远的节点)。这个名称来源于黑夜中半山腰迷路旅客的类比。假定这个旅客的营地在山顶，即使在黑夜中，这个旅客也知道每一点向上的步伐都是方向正确的一步。

只使用包含在航班安排的数据库中的信息，下面是将爬山启发信息整合到路线程序的方法：选择距离当前位置尽可能远的连接航班，以期望它能够距离目标比较近。为此，以 Search 的深度优先搜索版本开始，并修改 find() 例程，如下所示：

```
// Hill-climbing version.
// Given from, find the farthest away connection.
// Return true if a connection is found,
// and false otherwise.
bool Search::find(string from, FlightInfo &f)
{
    int pos = -1;
    int dist = 0;

    for(unsigned i=0; i < flights.size(); i++) {
        if(flights[i].from == from && !flights[i].skip) {
            // Use the longest flight.
            if(flights[i].distance > dist) {
                pos = i;
                dist = flights[i].distance;
            }
        }
    }

    if(pos != -1) {
        f = flights[pos];
        flights[pos].skip = true; // prevent reuse

        return true;
    }
}
```

```
    return false;
}
```

现在，find()例程搜索整个数据库，查找距离出发城市最远的连接。
为了清楚起见，整个爬山程序如下所示：

```
// Search for a connection by hill climbing.
#include <iostream>
#include <stack>
#include <string>
#include <vector>

using namespace std;

// Flight information.
struct FlightInfo {
    string from; // departure city
    string to; // destination city
    int distance; // distance between from and to
    bool skip; // used in backtracking

    FlightInfo() {
        from = "";
        to = "";
        distance = 0;
        skip = false;
    }

    FlightInfo(string f, string t, int d) {
        from = f;
        to = t;
        distance = d;
        skip = false;
    }
};

// This version of search finds connections
// through the heuristic of hill climbing.
class Search {
    // This vector holds the flight information.
    vector<FlightInfo> flights;

    // This stack is used for backtracking.
    stack<FlightInfo> btStack;

    // If there is a flight between from and to,
    // store the distance of the flight in dist.
    // Return true if the flight exists and
```

```

// false otherwise.
bool match(string from, string to, int &dist);

// Hill-climbing version.
// Given from, find the farthest away connection.
// Return true if a connection is found,
// and false otherwise.
bool find(string from, FlightInfo &f);

public:

// Put flights into the database.
void addflight(string from, string to, int dist) {
    flights.push_back(FlightInfo(from, to, dist));
}

// Show the route and total distance.
void route();

// Determine if there is a route between from and to.
void findroute(string from, string to);

// Return true if a route has been found.
bool routefound() {
    return btStack.size() != 0;
}
};

// Show the route and total distance.
void Search::route()
{
    stack<FlightInfo> rev;
    int dist = 0;
    FlightInfo f;

    // Reverse the stack to display route.
    while(!btStack.empty()) {
        f = btStack.top();
        rev.push(f);
        btStack.pop();
    }

    // Display the route.
    while(!rev.empty()) {
        f = rev.top();
        rev.pop();
        cout << f.from << " to ";
        dist += f.distance;
    }
}

```

```

    cout << f.to << endl;
    cout << "Distance is " << dist << endl;
}

// If there is a flight between from and to,
// store the distance of the flight in dist.
// Return true if the flight exists and
// false otherwise.
bool Search::match(string from, string to, int &dist)
{
    for(unsigned i=0; i < flights.size(); i++) {
        if(flights[i].from == from &&
            flights[i].to == to && !flights[i].skip)
        {
            flights[i].skip = true; // prevent reuse
            dist = flights[i].distance;
            return true;
        }
    }

    return false; // not found
}

// Hill-climbing version.
// Given from, find the farthest away connection.
// Return true if a connection is found,
// and false otherwise.
bool Search::find(string from, FlightInfo &f)
{
    int pos = -1;
    int dist = 0;

    for(unsigned i=0; i < flights.size(); i++) {
        if(flights[i].from == from && !flights[i].skip) {
            // Use the longest flight.
            if(flights[i].distance > dist) {
                pos = i;
                dist = flights[i].distance;
            }
        }
    }

    if(pos != -1) {
        f = flights[pos];
        flights[pos].skip = true; // prevent reuse

        return true;
    }
}

```

```

    return false;
}

// Determine if there is a route between from and to.
void Search::findroute(string from, string to)
{
    int dist;
    FlightInfo f;

    // See if at destination.
    if(match(from, to, dist)) {
        btStack.push(FlightInfo(from, to, dist));
        return;
    }

    // Try another connection.
    if(find(from, f)) {
        btStack.push(FlightInfo(from, to, f.distance));
        findroute(f.to, to);
    }
    else if(!btStack.empty()) {
        // Backtrack and try another connection.
        f = btStack.top();
        btStack.pop();
        findroute(f.from, f.to);
    }
}

int main() {
    char to[40], from[40];
    Search ob;

    // Add flight connections to database.
    ob.addflight("New York", "Chicago", 900);
    ob.addflight("Chicago", "Denver", 1000);
    ob.addflight("New York", "Toronto", 500);
    ob.addflight("New York", "Denver", 1800);
    ob.addflight("Toronto", "Calgary", 1700);
    ob.addflight("Toronto", "Los Angeles", 2500);
    ob.addflight("Toronto", "Chicago", 500);
    ob.addflight("Denver", "Urbana", 1000);
    ob.addflight("Denver", "Houston", 1000);
    ob.addflight("Houston", "Los Angeles", 1500);
    ob.addflight("Denver", "Los Angeles", 1000);

    // Get departure and destination cities.
    cout << "From? ";

```



```

cin.getline(from, 40);
cout << "To? ";

cin.getline(to, 40);

// See if there is a route between from and to.
ob.findroute(from, to);

// If there is a route, show it.
if(ob.routefound())
    ob.route();

return 0;
}

```

当程序运行时，得到的解决方案如下：

```

From? New York
To? Los Angeles
New York to Denver to Los Angeles
Distance is 2800

```

这个方案相当好。这条路线需要在路上停留的次数最少(只有一次)，并且这是最短的航线。因此，这个算法找到了可能是最好的航线。

然而，如果丹佛到洛杉矶的连接不存在，这个解决方案就不会如此完美。被找到的航线将会是从纽约到丹佛到休斯顿到洛杉矶，一共 4300 英里。在此情况下，这个解决方案爬上了一个“假山头”，因为到休斯顿的连接没有把我们带到距目标洛杉矶更近的地方。图 7-10 显示了第一个解决方案以及到“假山头”的路线。

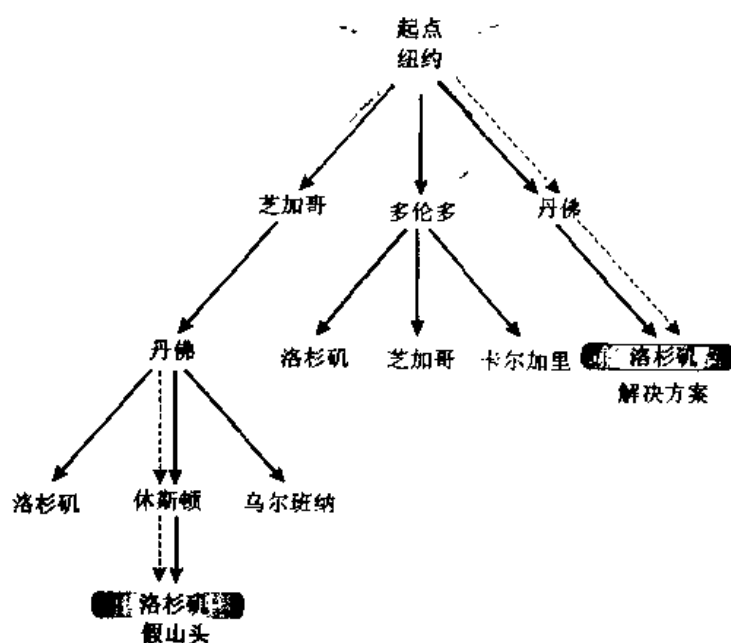


图 7-10 爬山法找到的解决方案的路径以及到假山头的路径

7.8.2 爬山法分析

在许多情况下，爬山法都能提供相当不错的解决方案，因为它试图降低在找到解决方案之前需要访问的节点的数量。然而，这个方法有三个缺点。首先是假山头的问题，如刚才所描述的那样。在此情况下，可能会导致很多的返回。第二个问题涉及到平稳状态，这是一种所有的下一步看上去都同样好(或者差)的情况。在此情况下，爬山法并不比深度优先搜索好。最后一个问题是山脊。在此情况下，爬山法的性能很差，因为在返回时，这个算法导致多次穿越山脊。尽管有这些潜在的问题，爬山法还是经常会增加找到好的解决方案的可能性。

7.9 最低成本搜索

与爬山搜索相对的是最低成本搜索。这种策略类似于穿着四轮旱冰鞋站在一座大山的路中间。毫无疑问您会觉得下山要比上山容易的多。换句话说，最低成本搜索法将采用阻力最小的路线。

将最低成本搜索法应用于航班安排问题，意味着将采用最短的连接航班，从而被找到的路线很有可能覆盖了最短的距离。不同于试图最小化连接数量的爬山法，最低成本搜索法试着最小化距离。

为了使用最低成本搜索，必须修改前面程序的 `find()` 函数，如下所示：

```
const int MAXDIST = 100000;

// Least-cost version.
// Given from, find the closest connection.
// Return true if a connection is found,
// and false otherwise.
bool Search::find(string from, FlightInfo &f)
{
    int pos = -1;
    int dist = MAXDIST; // longer than longest flight

    for(unsigned i=0; i < flights.size(); i++) {
        if(flights[i].from == from && !flights[i].skip) {
            // Use the shortest flight.
            if(flights[i].distance < dist) {
                pos = i;
                dist = flights[i].distance;
            }
        }
    }

    if(pos != -1) {
        f = flights[pos];
        flights[pos].skip = true; // prevent reuse

        return true;
    }

    return false;
}
```

使用这个版本的 find() 函数，找到的解决方案如下：

```
From? New York
To? Los Angeles
New York to Toronto to Los Angeles
Distance is 3000
```

正如您所看到的那样，这个搜索找到了一条好的航线，虽然不是最好的，但是可以接受。

图 7-11 显示了到达目标的最低成本路径。

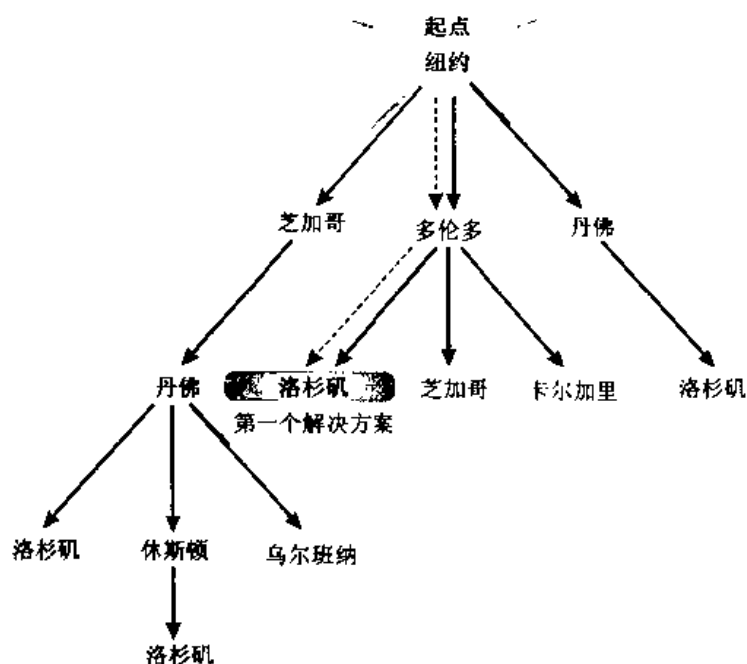


图 7-11 最低成本搜索法的解决方案路径

最低成本搜索分析

最低成本搜索和爬山法有着类似的优缺点，但刚好相反。在最低成本搜索中，可能有虚假的山谷、低地以及峡谷。在这个特殊的示例中，这个方法与爬山法运行的同样好。

7.10 寻找多解

有时寻找相同问题的几种解决方案是有意义的。这并不是寻找所有的解决方案(如同穷举搜索那样)。相反，多解提供了搜索空间中解决方案的代表性样本。

有多种方法生成多解，但是在此只使用两种方法。第一种方法是路径删除，第二种方法是节点删除。正如其名称所显示的那样，生成没有冗余的多解要求将已经发现的解决方案从系统中删除。记住，所有的这些方法都没有试图寻找全部的解决方案。寻找全部的解决方案是另一个问题，通常不会这么尝试，因为这样做意味着穷举搜索。

7.10.1 路径删除

生成多解的路径删除方法将当前解决方案中的所有节点都从数据库中删除，然后尝试寻找另一种解决方案。基本上，路径删除是从树上剪除分支。为了使用路径删除来寻找多解，您需要修改 `main()` 函数，如下所示：

```
// Path removal version.
int main() {
    char to[40], from[40];
    Search ob;

    // Add flight connections to database.
    ob.addflight("New York", "Chicago", 900);
    ob.addflight("Chicago", "Denver", 1000);
    ob.addflight("New York", "Toronto", 500);
    ob.addflight("New York", "Denver", 1800);
    ob.addflight("Toronto", "Calgary", 1700);
    ob.addflight("Toronto", "Los Angeles", 2500);
    ob.addflight("Toronto", "Chicago", 500);
    ob.addflight("Denver", "Urbana", 1000);
    ob.addflight("Denver", "Houston", 1000);
    ob.addflight("Houston", "Los Angeles", 1500);
    ob.addflight("Denver", "Los Angeles", 1000);

    // Get departure and destination cities.
    cout << "From? ";

    cin.getline(from, 40);
    cout << "To? ";

    cin.getline(to, 40);

    // Find multiple solutions.
    for(;;) {
        // See if there is a connection.
        ob.findroute(from, to);

        // If no new route was found, then end.
        if(!ob.routeFound()) break;

        ob.route();
    }

    return 0;
}
```

在此，加入了一个 `for` 循环，这个循环一直迭代，直到返回堆栈为空。当返回堆栈为空时，不会发现解决方案(在此情况下，没有另外的解决方案)。不需要进行另外的修改，因为作为这个解决方案一部分的任何连接都会使得 `skip` 字段被标记。因此，这种连接再也不会被 `find()` 函数发现，也不会是下一个解决方案的一部分；它不会被再次发现。

如果您使用了 Search 的原始深度优先版本(本章一开始所示)以及前面的 main()函数, 会找到如下的解决方案:

```
From? New York
To? Los Angeles
New York to Chicago to Denver to Los Angeles
Distance is 2900
New York to Toronto to Los Angeles
Distance is 3000
New York to Denver to Houston to Los Angeles
Distance is 4300
```

这个搜索找到了 3 个解决方案。然而需要注意, 它们都不是最好的解决方案。

7.10.2 节点删除

第二种强制生成另外的解决方案的方法是节点删除法, 这种方法只是简单地删除当前解决方案的最后一个节点, 并再次尝试。为此, 需要修改 main()函数, 从航班数据库中删除最后一个连接, 清除所有的 skip 字段, 并获取一个新的空堆栈以保存下一个解决方案。更新的 main()函数如下所示:

```
// Node removal version
int main() {
    char to[40], from[40];
    Search ob;
    FlightInfo f;

    // Add flight connections to database.
    ob.addflight("New York", "Chicago", 900);
    ob.addflight("Chicago", "Denver", 1000);
    ob.addflight("New York", "Toronto", 500);
    ob.addflight("New York", "Denver", 1800);
    ob.addflight("Toronto", "Calgary", 1700);
    ob.addflight("Toronto", "Los Angeles", 2500);
    ob.addflight("Toronto", "Chicago", 500);
    ob.addflight("Denver", "Urbana", 1000);
    ob.addflight("Denver", "Houston", 1000);
    ob.addflight("Houston", "Los Angeles", 1500);
    ob.addflight("Denver", "Los Angeles", 1000);

    // Get departure and destination cities.
    cout << "From? ";

    cin.getline(from, 40);
    cout << "To? ";

    cin.getline(to, 40);

    // Find multiple solutions.
    for(;;) {
        // See if there is a connection.
```

```

    ob.findroute(from, to);

    // If no new route was found, then end.
    if(!ob.routeFound()) break;

    // Save the flight on top-of-stack.
    f = ob.getTOS();

    ob.route(); // display the current route.

    ob.resetAllSkip(); // reset the skip fields

    // Remove last flight in previous solution
    // from the flight database.
    ob.remove(f);
}

return 0;
}

```

为了从航班数据库中删除前面解决方案的最后一个连接, `main()` 函数首先调用 `getTOS()` 来获取这个连接, `getTOS()` 是在 `Search` 类中声明的内联函数, 如下所示:

```

// Return flight on top of stack.
FlightInfo getTOS() {
    return btStack.top();
}

```

这个函数返回回溯堆栈的栈顶处的航班信息, 这是路线中的最后一个连接。为了删除这个连接, `main()` 函数调用 `remove()` 函数, 如下所示:

```

// Remove a connection.
void Search::remove(FlightInfo f) {
    for(unsigned i=0; i< flights.size(); i++)
        if(flights[i].from == f.from &&
            flights[i].to == f.to)
            flights[i].from = "";
}

```

通过将出发城市的名称赋予长度为 0 的字符串, 完成了对这个连接的删除。为了清除 `skip` 字段, `main()` 函数调用 `resetAllSkip()`, 如下所示:

```

// Reset all skip fields.
void Search::resetAllSkip() {
    for(unsigned i=0; i< flights.size(); i++)
        flights[i].skip = false;
}

```

这个函数只是简单地将 `skip` 字段设置为 `false`。(记住将函数 `resetAllSkip()` 和 `remove()` 的原型加入到 `Search` 类的声明中)。

由于需要这么多的修改, 为了清楚起见, 在此显示了全部的节点删除程序。注意它也使用

了 Search 的原始深度优先版本。

```
// Search for multiple routes by use of node removal.
#include <iostream>
#include <stack>
#include <string>
#include <vector>

using namespace std;

// Flight information.
struct FlightInfo {
    string from;    // departure city
    string to;      // destination city
    int distance;   // distance between from and to
    bool skip;      // used in backtracking

    FlightInfo() {
        from = "";
        to = "";
        distance = 0;
        skip = false;
    }

    FlightInfo(string f, string t, int d) {
        from = f;
        to = t;
        distance = d;
        skip = false;
    }
};

// Find multiple solutions via node removal.
class Search {
    // This vector holds the flight information.
    vector<FlightInfo> flights;

    // This stack is used for backtracking.
    stack<FlightInfo> btStack;

    // If there is a flight between from and to,
    // store the distance of the flight in dist.
    // Return true if the flight exists and
    // false otherwise.
    bool match(string from, string to, int &dist);

    // Given from, find any connection.
    // Return true if a connection is found,
    // and false otherwise.
    bool find(string from, FlightInfo &f);

public:
```

```

// Put flights into the database.
void addflight(string from, string to, int dist) {
    flights.push_back(FlightInfo(from, to, dist));
}

// Show the route and total distance.
void route();

// Determine if there is a route between from and to.
void findroute(string from, string to);

// Return true if a route has been found.
bool routefound() {
    return btStack.size() != 0;
}

// Return flight on top of stack.
FlightInfo getTOS() {
    return btStack.top();
}

// Reset all skip fields.
void resetAllSkip();

// Remove a connection.
void remove(FlightInfo f);
};

// Show the route and total distance.
void Search::route()
{
    stack<FlightInfo> rev;
    int dist = 0;
    FlightInfo f;

    // Reverse the stack to display route.
    while(!btStack.empty()) {
        f = btStack.top();
        rev.push(f);
        btStack.pop();
    }

    // Display the route.
    while(!rev.empty()) {
        f = rev.top();
        rev.pop();
        cout << f.from << " to ";
        dist += f.distance;
    }

    cout << f.to << endl;
}

```



```

    cout << "Distance is " << dist << endl;
}

// If there is a flight between from and to,
// store the distance of the flight in dist.
// Return true if the flight exists and
// false otherwise.
bool Search::match(string from, string to, int &dist)
{
    for(unsigned i=0; i < flights.size(); i++) {
        if(flights[i].from == from &&
            flights[i].to == to && !flights[i].skip)
        {
            flights[i].skip = true; // prevent reuse
            dist = flights[i].distance;
            return true;
        }
    }

    return false; // not found
}

// Given from, find any connection.
// Return true if a connection is found,
// and false otherwise.
bool Search::find(string from, FlightInfo &f)
{
    for(unsigned i=0; i < flights.size(); i++) {
        if(flights[i].from == from && !flights[i].skip) {
            f = flights[i];
            flights[i].skip = true; // prevent reuse

            return true;
        }
    }

    return false;
}

// Determine if there is a route between from and to.
void Search::findroute(string from, string to)
{
    int dist;
    FlightInfo f;

    // See if at destination.
    if(match(from, to, dist)) {
        btStack.push(FlightInfo(from, to, dist));
        return;
    }

    // Try another connection.

```

```

if(find(from, f)) {
    btStack.push(FlightInfo(from, to, f.distance));
    findroute(f.to, to);
}
else if(!btStack.empty()) {
    // Backtrack and try another connection.
    f = btStack.top();
    btStack.pop();
    findroute(f.from, f.to);
}
}

// Reset all skip fields.
void Search::resetAllSkip() {
    for(unsigned i=0; i< flights.size(); i++)
        flights[i].skip = false;
}

// Remove a connection.
void Search::remove(FlightInfo f) {
    for(unsigned i=0; i< flights.size(); i++)
        if(flights[i].from == f.from &&
            flights[i].to == f.to)
            flights[i].from = "";
}

// Node removal version.
int main() {
    char to[40], from[40];
    Search ob;
    FlightInfo f;

    // Add flight connections to database.
    ob.addflight("New York", "Chicago", 900);
    ob.addflight("Chicago", "Denver", 1000);
    ob.addflight("New York", "Toronto", 500);
    ob.addflight("New York", "Denver", 1800);
    ob.addflight("Toronto", "Calgary", 1700);
    ob.addflight("Toronto", "Los Angeles", 2500);
    ob.addflight("Toronto", "Chicago", 500);
    ob.addflight("Denver", "Urbana", 1000);
    ob.addflight("Denver", "Houston", 1000);
    ob.addflight("Houston", "Los Angeles", 1500);
    ob.addflight("Denver", "Los Angeles", 1000);

    // Get departure and destination cities.
    cout << "From? ";

    cin.getline(from, 40);
    cout << "To? ";

    cin.getline(to, 40);

```

```

// Find multiple solutions.
for(;;) {
// See if there is a connection.
ob.findroute(from, to);

// If no new route was found, then end.
if(!ob.routeFound()) break;

// Save the flight on top-of-stack.
f = ob.getTOS();

ob.route(); // display the current route.

ob.resetAllSkip(); // reset the skip fields

// Remove last flight in previous solution
// from the flight database.
ob.remove(f);
}

return 0;
}

```

这个程序找到了如下的路线：

```

From? New York
To? Los Angeles
New York to Chicago to Denver to Los Angeles
Distance is 2900
New York to Chicago to Denver to Houston to Los Angeles
Distance is 4400
New York to Toronto to Los Angeles
Distance is 3000

```

在此情况下，第二个解决方案可能是最差的路线，但是找到了两个相当好的解决方案。注意节点删除法找到的这组解决方案与路径删除法找到的方案不同。不同的生成多解的方法通常会产生不同的结果。

7.11 寻找“最优”解决方案

所有前面的搜索技术都只关心寻找一个解决方案——任何解决方案都可以。当您观察启发式搜索时，会努力增加找到好的解决方案的可能性。但是却没有试图确保寻找最优解决方案。然而，有时您只想要最优解决方案。记住，这里所讲的“最优”，只是意味着使用一个生成多解的技术所能找到的最好的路线——实际上它可能不是最好的解决方案(当然，寻找最好的解决方案需要花费大量时间的穷举搜索)

在抛开很好使用的航班安排示例之前，考虑一个寻找给定约束下最优航线的程序，这个约束要求距离最小化。为此，这个程序使用了路径删除方法来生成多解，并使用最低成本法搜索

最小距离。寻找最短航线的关键是使得某个解决方案比前面生成的解决方案的航线短。当不再产生解决方案时，最优解决方案就确定了。

全部的“最优解决方案”程序如下所示。注意这个程序创建了一个附加的堆栈，名为 `optimal`，这个堆栈保存最优解，还有一个名为 `minDist` 的实例变量，用这个变量来跟踪距离。`route()`和 `main()`也有少量的修改。

```
// Find an "optimal" solution using least-cost with path removal.
#include <iostream>
#include <stack>
#include <string>
#include <vector>

using namespace std;

// Flight information.
struct FlightInfo {
    string from; // departure city
    string to; // destination city
    int distance; // distance between from and to
    bool skip; // used in backtracking

    FlightInfo() {
        from = "";
        to = "";
        distance = 0;
        skip = false;
    }

    FlightInfo(string f, string t, int d) {
        from = f;
        to = t;
        distance = d;
        skip = false;
    }
};

const int MAXDIST = 100000;

// Find connections using least cost.
class Optimal {
    // This vector holds the flight information.
    vector<FlightInfo> flights;

    // This stack is used for backtracking.
    stack<FlightInfo> btStack;

    // This stack holds the optimal solution.
    stack<FlightInfo> optimal;

    int minDist;
```

```

// If there is a flight between from and to,
// store the distance of the flight in dist.
// Return true if the flight exists and
// false otherwise.
bool match(string from, string to, int &dist);

// Least-cost version.
// Given from, find the closest connection.
// Return true if a connection is found,
// and false otherwise.
bool find(string from, FlightInfo &f);

public:

// Constructor
Optimal() {
    minDist = MAXDIST;
}

// Put flights into the database.
void addflight(string from, string to, int dist) {
    flights.push_back(FlightInfo(from, to, dist));
}

// Show the route and total distance.
void route();

// Display the optimal route.
void Optimal::showOpt();

// Determine if there is a route between from and to.
void findroute(string from, string to);

// Return true if a route has been found.
bool routefound() {
    return btStack.size() != 0;
}
};

// Show the route and total distance.
void Optimal::route()
{
    stack<FlightInfo> optTemp;
    int dist = 0;
    FlightInfo f;

    // Reverse the stack to display route.
    while(!btStack.empty()) {
        f = btStack.top();
        optTemp.push(f);
        btStack.pop();
        dist += f.distance;
    }
}

```

```

    }

    // If shorter, keep this route.
    if(minDist > dist) {
        optimal = optTemp;
        minDist = dist;
    }
}

// Display the optimal route.
void Optimal::showOpt()
{
    FlightInfo f;
    int dist = 0;

    cout << "Optimal solution is:\n";
    // Display the optimal route.
    while(!optimal.empty()) {
        f = optimal.top();
        optimal.pop();
        cout << f.from << " to ";
        dist += f.distance;
    }

    cout << f.to << endl;
    cout << "Distance is " << dist << endl;
}

// If there is a flight between from and to,
// store the distance of the flight in dist.
// Return true if the flight exists and
// false otherwise.
bool Optimal::match(string from, string to, int &dist)
{
    for(unsigned i=0; i < flights.size(); i++) {
        if(flights[i].from == from &&
           flights[i].to == to && !flights[i].skip)
        {
            flights[i].skip = true; // prevent reuse
            dist = flights[i].distance;
            return true;
        }
    }

    return false; // not found
}

// Least-cost version.
// Given from, find the closest connection.
// Return true if a connection is found,
// and false otherwise.
bool Optimal::find(string from, FlightInfo &f)

```

```

{
    int pos = -1;
    int dist = MAXDIST; // longer than longest flight

    for(unsigned i=0; i < flights.size(); i++) {
        if(flights[i].from == from && !flights[i].skip)
        {
            // Use the shortest flight.
            if(flights[i].distance < dist) {
                pos = i;
                dist = flights[i].distance;
            }
        }
    }

    if(pos != -1) {
        f = flights[pos];
        flights[pos].skip = true; // prevent reuse

        return true;
    }

    return false;
}

// Determine if there is a route between from and to.
void Optimal::findroute(string from, string to)
{
    int dist;
    FlightInfo f;

    // See if at destination.
    if(match(from, to, dist)) {
        btStack.push(FlightInfo(from, to, dist));
        return;
    }

    // Try another connection.
    if(find(from, f)) {
        btStack.push(FlightInfo(from, to, f.distance));
        findroute(f.to, to);
    }
    else if(!btStack.empty()) {
        // Backtrack and try another connection.
        f = btStack.top();
        btStack.pop();
        findroute(f.from, f.to);
    }
}

// Find "optimal" solution by using least-cost with path removal.
int main() {

```

```

char to[40], from[40];
Optimal ob;

// Add flight connections to database.
ob.addflight("New York", "Chicago", 900);
ob.addflight("Chicago", "Denver", 1000);
ob.addflight("New York", "Toronto", 500);
ob.addflight("New York", "Denver", 1800);
ob.addflight("Toronto", "Calgary", 1700);
ob.addflight("Toronto", "Los Angeles", 2500);
ob.addflight("Toronto", "Chicago", 500);
ob.addflight("Denver", "Urbana", 1000);
ob.addflight("Denver", "Houston", 1000);
ob.addflight("Houston", "Los Angeles", 1500);
ob.addflight("Denver", "Los Angeles", 1000);

// Get departure and destination cities.
cout << "From? ";

cin.getline(from, 40);
cout << "To? ";

cin.getline(to, 40);

// Find multiple solutions.
for(;;) {
    // See if there is a connection.
    ob.findroute(from, to);

    // If no route found, then end.
    if(!ob.routefound()) break;

    ob.route();
}

// Display optimal solution.
ob.showOpt();

return 0;
}

```

这个程序的输出如下所示:

```

From? New York
To? Los Angeles
Optimal solution is:
New York to Chicago to Denver to Los Angeles
Distance is 2900

```

在此情况下, 这个“最优”解决方案并不是最好的, 但是它仍然是非常好的一个解决方案。如前所述, 当使用基于 AI 的搜索时, 某种搜索技术所能找到的最佳解决方案并不总是可能存

在那个最佳方案。在前面的程序中，您或许想使用其他的搜索技术，来观察会找到什么样的“最优”解决方案。

前面的方法中有一个不足，即所有的路径都通向终点。改进的方法在长度等于或者大于当前最小长度时，就会停止跟踪这条路径。您可能需要修改这个程序来适应这种增强。

7.12 回到丢失钥匙的问题

为了总结本章关于问题的解决方案，提供一个搜索本章前面提到的丢失钥匙的 C++ 程序比较合适。下面的代码使用了与寻找两个城市间航线问题中相同的方法，因此没有对这个程序进行更多的解释：

```
// Search for the lost keys.
#include <iostream>
#include <stack>
#include <string>
#include <vector>

using namespace std;

// Room information.
struct RoomInfo {
    string from;
    string to;
    bool skip;

    RoomInfo() {
        from = "";
        to = "";
        skip = false;
    }

    RoomInfo(string f, string t) {
        from = f;
        to = t;
        skip = false;
    }
};

// Find the keys using a depth-first search.
class Search {
    // This vector holds the room information.
    vector<RoomInfo> rooms;

    // This stack is used for backtracking.
    stack<RoomInfo> btStack;

    // Return true if a path exists between
    // from and to. Return false otherwise.
    bool match(string from, string to);
};
```

```

// Given from, find any path.
// Return true if a path is found,
// and false otherwise.
bool find(string from, RoomInfo &f);

public:

// Put rooms into the database.
void addroom(string from, string to)
{
    rooms.push_back(RoomInfo(from, to));
}

// Show the route taken.
void route();

// Determine if there is a path between from and to.
void findkeys(string from, string to);

// Return true if the keys have been found.
bool keysfound() {
    return !btStack.empty();
}
};

// Show the route.
void Search::route()
{
    stack<RoomInfo> rev;
    RoomInfo f;

    // Reverse the stack to display route.
    while(!btStack.empty()) {
        f = btStack.top();
        rev.push(f);
        btStack.pop();
    }

    // Display the route.
    while(!rev.empty()) {
        f = rev.top();
        rev.pop();
        cout << f.from << " to ";
    }

    cout << f.to << endl;
}

// Return true if a path exists between
// from and to. Return false otherwise.
bool Search::match(string from, string to)

```

```

{
    for(unsigned i=0; i < rooms.size(); i++) {
        if(rooms[i].from == from &&
           rooms[i].to == to && !rooms[i].skip)
        {
            rooms[i].skip = true; // prevent reuse
            return true;
        }
    }

    return false; // not found
}

// Given from, find any path.
// Return true if a path is found,
// and false otherwise.
bool Search::find(string from, RoomInfo &f)
{
    for(unsigned i=0; i < rooms.size(); i++) {
        if(rooms[i].from == from && !rooms[i].skip) {
            f = rooms[i];
            rooms[i].skip = true; // prevent reuse

            return true;
        }
    }

    return false;
}

// Find the keys.
void Search::findkeys(string from, string to)
{
    RoomInfo f;

    // See if keys are found.
    if(match(from, to)) {
        btStack.push(RoomInfo(from, to));
        return;
    }

    // Try another room.
    if(find(from, f)) {
        btStack.push(RoomInfo(from, to));
        findkeys(f.to, to);
    }
    else if(!btStack.empty()) {
        // Backtrack and try another path.
        f = btStack.top();
        btStack.pop();
        findkeys(f.from, f.to);
    }
}

```

```

}

int main() {
    Search ob;

    // Add rooms to database.
    ob.addroom("front_door", "lr");
    ob.addroom("lr", "bath");
    ob.addroom("lr", "hall");
    ob.addroom("hall", "bd1");
    ob.addroom("hall", "bd2");
    ob.addroom("hall", "mb");
    ob.addroom("lr", "kitchen");
    ob.addroom("kitchen", "keys");

    // Find the keys.
    ob.findkeys("front_door", "keys");

    // If keys are found, show the path.
    if(ob.keysfound())
        ob.route();

    return 0;
}

```

7.13 尝试完成以下任务

由于基于 AI 的搜索是一个挑战，并且仍然是计算机科学中具有挑战性的一个领域，因此对它的体验充满了乐趣。在此有一些您可以尝试完成的任务。首先，用试探搜索方法替代广度优先方法并观察结果。其次，在寻找多解时，或者在寻找“最优”解决方案时，替代广度优先搜索方法。最后，试着处理其他现实生活中的搜索问题。

第 8 章 定制 STL 容器

本书的一个中心主题是 C++ 提供给程序员的原始功能。或许标准模板库(STL)最能说明这个问题，这个库改变了程序员编写程序的方式。STL 包含了一组成熟的模板类和函数，这些类和函数实现了常用的数据结构和算法。对涉及到数据存储及获取的各种编程问题，STL 提供了可以直接使用的解决方案，因此基于 STL 的代码变得越来越普遍。例如，第 2 章的垃圾回收子系统和第 3 章的线程控制面板都使用了 STL，从而极大地简化了代码。在过去，如果需要某个数据结构，如链表、堆栈或者队列，必须手工编码。现在，程序员可以简单地使用 STL 提供的某个解决方案。

STL 的核心是容器。容器是一个保存其他对象的对象。STL 提供了几种内建的容器以支持如堆栈、队列、链表以及向量之类的数据结构。由于容器是模板化的，因此能够存储任何类型的对象，包括您创建的类对象。

尽管内建的容器非常有用，但是您并不会受到它们的限制。STL 最引人注目的特征是允许您创建自己的容器。因此，STL 是可扩展的。一旦您创建了一个容器，它就自动与 STL 的其余部分完全兼容。

在这一章，您将会看到如何创建定制的 STL 容器。尽管定制容器并不困难，但是许多程序员开始时对这种想法还是有点恐惧。原因之一是 STL 基于模板的语法乍看上去是无法征服的。事实上，STL 是概念上非常整洁的、易于理解的子系统。如果您遵循几个规则，将毫无问题地创建您自己的容器。

本章开发的定制容器实现了一个名为 `RangeArray` 的范围可以选择的动态数组。当使用 `RangeArray` 时，您可以指定开始和结束的索引。例如，`RangeArray` 允许创建一个从 -10 到 10 的数组。

8.1 STL 的简要回顾

STL 是一个很大的话题，对其详细分析需要上百页的篇幅，而且对 STL 的完整分析远远超出了本书的范围。因此，本章假定您具有 STL 的基本知识。在此给出了 STL 的主要术语及其组成部分，STL 基于 3 个主要的特性：容器、迭代器以及算法。如前所述，容器是包含其他对象的对象；迭代器是类似于指针的对象，通过迭代器，可以使用类似于用指针遍历数组的方法来遍历容器；算法是对容器中元素的操作，如修改、复制或者处理其他元素。

除了容器、迭代器以及算法，STL 还需要其他一些标准组件的支持，如分配器、适配器、谓词以及函数对象。下面部分进一步介绍主要的 STL 要素。

提示：

对于 STL 的完整讨论，作者推荐参考他的另一本书 *STL Programming From the Ground Up*, McGraw - Hill/Osborne。

8.1.1 容器

STL 定义了两类容器：序列式容器和关联式容器。序列式容器拥有一个线性的对象链表。STL 提供了几种内建的序列式容器，包括 `vector` (定义了动态数组)、`deque` (创建了双端队列) 以及 `list` (实现了一个双向链表)。关联式容器存储关键字/值对。因此，关联式容器允许基于关键字来高效地检索值。例如，`map` 可通过唯一的关键词访问值。因此，`map` 存储的关键词/值对，并允许检索值和对应的关键词。

每个容器类定义一组可能应用于容器的函数。例如，`list` 容器包含插入、删除和合并元素的函数。`stack` 包含压入和弹出值的函数。

8.1.2 算法

算法作用于容器。算法定义了对容器的内容进行操作的方法。其功能包括对容器内容的初始化、存储、搜索以及转换。许多算法只是针对容器中某个范围的元素而不是整个容器操作。算法的示例包括 `copy()`、`remove()`、`replace()` 以及 `find()`。

8.1.3 迭代器

迭代器是类似于指针的对象，可以将其增加或者减小。您还能够对其应用 * 运算符。迭代器由不同容器定义的 `iterator` 类型声明。

STL 还支持反向迭代器。反向迭代器以相反的方向遍历一个序列。因此，如果反向迭代器指向某个序列的结尾，则增加这个迭代器会使得它指向结尾的前一个元素。

8.2 其他的 STL 实体

除了容器、迭代器以及算法之外，其他一些 STL 的元素也扮演着重要的角色：分配器、函数对象、谓词、适配器、绑定器以及否定器 (`negator`)。

每一个容器都定义了一个分配器。分配器为容器管理内存的分配。默认的分配器是 `allocator` 类的一个对象，但是，如果特定的应用程序需要，就可以定义自己的分配器。在多数情况下，默认的分配器就足够了。

函数对象是定义 `operator()` 的类。在此有几个预定义的函数对象，如 `less()`、`greater()`、`plus()`、`minus()`、`multiplies()` 以及 `divides()`。或许使用最广泛的函数对象是 `less()`，这个函数用来判断一个对象什么时候小于另一个对象。当使用 STL 算法时，函数对象可以用来代替函数指针。

就最普遍的意义来讲，适配器将一个对象转换为另一个对象。适配器分为容器适配器、迭代器适配器以及函数适配器。`queue` 是容器适配器的一个示例，它使得 `deque` 容器可以像标准队列那样使用。适配器简化了许多复杂的情况。

有些算法和容器使用了一种特殊类型的称为谓词的函数。在此有两种版本的谓词：一元谓词和二元谓词。一元谓词具有一个参数，二元谓词具有两个参数。这些函数返回 `true/false`，但是它们返回 `true` 或 `false` 的准确条件是由您定义的。某些算法使用了特定类型的二元谓词来比较两个元素。如果第一个参数小于第二个参数，这些比较函数则返回 `true`。

在 STL 中常用的其他两个实体是绑定器和函数对象。绑定器将参数与函数对象绑定。STL 定义了两种绑定器：`bind2nd()` 和 `bind1st()`。否定器返回谓词的对立值。在此有两个否定器：`not1()`

和 not2()。绑定器和否定器增加了 STL 功能的多样性。

8.3 定制容器的要求

在设计一个定制的容器之前，有必要准确知道它必须提供的功能。根据标准 C++，所有的容器都必须提供指定的一组类型、运算符以及成员函数。除了这些常用的元素之外，序列式容器和关联式容器都加入了一些特定的要求。如果您遵循这些规则，您定制的容器就可以被 STL 定义的所有其他实体(包括分配器、函数对象、谓词以及绑定器)使用。设计良好的定制容器可以完全整合到 STL 中。下面部分描述了对于容器的要求。

8.3.1 一般要求

每一种容器(序列式容器或者关联式容器)都必须通过分配器而不是 new 和 delete 来管理内存。因此，容器必须使用分配器的成员函数来分配并释放内存。当容器创建时，分配器作为一个参数被指定。然而，C++ 提供了一个默认的分配器类，名为 allocator，对于 allocator 类型的对象，allocator 参数可以采用默认值，所有的标准容器都是如此。所有的分配器都必须提供与 allocator 相同的成员函数(当然，可以有不同的实现)。通过这种方法，容器可以使用程序员提供的任何分配器。本章容器使用的分配器函数如表 8-1 所示。

表 8-1 本章使用的分配器函数

函 数	描 述
pointer allocate(size_type num,typename allocator<void>::const_pointer h=0);	返回一个指向已分配内存的指针，这块内存足够大，能够保存 T 类型的 num 个对象。h 的值是一个对函数的提示，该函数可以用来满足或者忽略请求
void construct(pointer ptr,const_reference val);	在 ptr 处构建一个 T 类型的对象
void deallocate(pointer ptr,size_type num);	释放在 ptr 处开始的 T 类型的 num 个对象。ptr 的值必须已经从 allocate() 获取
void destroy(pointer ptr);	销毁 ptr 处的对象。会自动调用对象的析构函数
size_type max_size()const throw();	返回可分配的 T 类型对象的最大数量

每个容器都必须提供以下这些类型：

```
iterator      const_iterator      reference      const_reference
value_type    size_type            difference_type
```

可反转的容器(支持双向迭代器)必须提供以下这些类型：

```
reverse_iterator      const_reverse_iterator
```

所有的容器都必须提供默认的构造函数和复制构造函数，默认的构造函数创建了一个长度为 0 的容器。此外还需要不同参数的构造函数，序列式容器和关联式容器的具体形式有所不同。

另外还需要析构函数。

必须支持下面的成员函数：

<code>begin()</code>	<code>clear()</code>	<code>empty()</code>	<code>end()</code>
<code>erase()</code>	<code>insert()</code>	<code>max_size()</code>	<code>rbegin()</code>
<code>rend()</code>	<code>size()</code>	<code>swap()</code>	

当然，只有可反转容器需要 `rbegin()` 和 `rend()` 函数。一些函数具有重载的形式。

提示：

所有 STL 内建的容器都提供函数 `get_allocator()`，但是这对于定制的容器并不需要。

容器必须定义迭代器函数(如 `begin()`)的事实说明，容器必须支持所有需要的迭代器操作。所有的容器都必须支持下面的运算符：

<code>=</code>	<code>==</code>	<code>!=</code>	<code>></code>
<code><</code>	<code><=</code>	<code>>=</code>	

8.3.2 序列式容器的其他要求

除了默认的构造函数和复制构造函数之外，序列式容器必须提供构造函数来创建并初始化指定数量的元素。并且还必须提供构造函数来创建并初始化给定元素范围的对象。因此，必须支持如下形式的构造函数：

```
Cnt( )
Cnt(c)
Cnt(num, val)
Cnt(start, end)
```

在此，`c` 是 `Cnt` 类型的对象，`num` 是指定一个数量的整型数，`val` 是与存储在 `Cnt` 中的对象类型兼容的值，`start` 和 `end` 是用来初始化容器的元素范围的迭代器。定制的容器可以指定其他的构造函数。

提示：

除了复制构造函数之外，STL 序列式容器内建的构造函数接受一个参数来指定分配器(默认为 `allocator`)，但并不要求这么做。

标准 C++ 为序列式容器定义了如下可选的成员函数：

<code>at()</code>	<code>back()</code>	<code>front()</code>	<code>pop_back()</code>
<code>pop_front()</code>	<code>push_back()</code>	<code>push_front()</code>	

下标运算符 `[]` 也是可选的。当然，您可以加入自己设计的其他成员函数。

8.3.3 关联式容器的要求

所有的关联式容器必须定义这些附加的类型：

<code>key_compare</code>	<code>key_type</code>	<code>value_compare</code>
--------------------------	-----------------------	----------------------------

除了默认构造函数和复制构造函数之外，所有的关联式容器必须提供允许您指定比较函数的构造函数。您还必须定义一个构造函数，用来创建并初始化给定元素范围的对象。这种构造函数的版本之一必须使用默认的比较函数。其他的版本必须允许用户指定比较函数。也就是说，必须支持如下格式的构造函数：

```
Cnt( )
Cnt(c)
Cnt(comp)
Cnt(start, end)
Cnt(start, end, comp)
```

在此，`c` 是 `Cnt` 类型的对象，`start` 和 `end` 是用来初始化容器的元素范围的迭代器，`comp` 是比较函数。定制容器可以指定另外的构造函数。

提示：

除了复制构造函数之外，STL 关联式容器内建的构造函数接受了一个参数来指定分配器 (allocator 默认的)，但并不要求这么做。

关联式容器必须提供这些附加的成员函数：

```
count( )           equal_range( )       find( )           key_comp( )
lower_bound( )     upper_bound( )       value_comp( )
```

8.4 创建范围可选的动态数组容器

本章的剩余部分开发了一个定制的序列式容器，名为 `RangeArray`，这个容器提供了一个范围可选的动态数组。尽管这里所显示的示例演示的是序列式容器的创建，然而其中大多数的概念也可以让用户来实现关联式容器。

8.4.1 `RangeArray` 的运行方式

正如您所知道的那样，在 C++ 中，所有的数组都从 0 开始，并且不允许负的索引。然而，允许程序员指定不同端点的数组对于某些应用程序是有意义的。例如笛卡儿坐标平面：每一个数轴都是具有正值和负值的直线。在程序中代表这种直线的简洁方法是使用允许正负索引的数组。例如，给定一条从 -5 到 5 的线段，您可能会使用能够以如下方式索引的数组：

-5	-4	-3	-2	-1	0	1	2	3	4	5
----	----	----	----	----	---	---	---	---	---	---

在此开发的 `RangeArray` 容器可以创建这样的数组。更为常见的是，`RangeArray` 允许为数组索引指定任意的上界和下界。

`RangeArray` 是一个动态容器，允许数组在正负两个方向增长。在这点上，它类似于内建的容器 `vector`。`RangeArray` 支持序列式容器要求的所有操作，加上可选的 `[]` 运算符，以及可选的函数 `at()`、`push_front()`、`pop_front()` 等。

下面给出了使用 `RangeArray` 的示例程序代码：

```
// This creates an array that runs from -3 to 4
// and is initialized to zero.
RangeArray<int> ob(-3, 4, 0);

// Load the values -3 to 4 into ob.
for(int i = -3; i < 5; i++) ob[i] = i;
// ...
cout << ob[-2];
// ...
ob[2] = ob[-1] % 2;
```

您可能已经猜到了，第一行创建了一个从-3到4的 `RangeArray` 对象，数组中的每个元素都被初始化为0。其余的行说明这个数组可以通过从-3到4的索引来访问。

通常，`RangeArray` 允许通过指定数组的上下界以及用来初始化数组中每个元素的值来创建一个对象。也就是说，`RangeArray` 支持如下的构造函数：

```
RangeArray(lowerbound, upperbound, initvalue)
```

一旦创建了这样的数组，就可以使用其范围内的任何值来对其索引。这意味着允许任意的索引，包括负的索引。

由于 `RangeArray` 是动态的，因此它允许插入或者删除元素。当发生这样的操作时，数组需要增长或者收缩。然而，关键是数组可以在两个方向增长或者收缩。如果在负方向加入元素，则负方向增长。如果正方向的元素被删除，则正方向收缩。

在开始之前有必要指出，有意没有对 `RangeArray` 容器的实现进行高性能优化。相反，以最清晰的方式将它优化。这样做的目的是清楚地显示创建定制容器时需要的步骤。同样，如此设计也考虑到了理解的容易程度以及实现的直接性。

8.4.2 完整的 `RangeArray` 类

在此首先给出 `RangeArray` 的全部代码。您将会发现，在创建您自己的容器类时，即使最简单的容器也会变为相当大的类。原因当然是必须满足一些要求。尽管单个的要求都不困难，但是将它们组合到一起就需要相当多的代码。不要被吓倒，在下面部分将逐步介绍每一个部分：

```
// A custom container that implements a
// range-selectable array.
//
// Call this file ra.h
//
#include <iostream>
#include <iterator>
#include <algorithm>
#include <cstdlib>
#include <stdexcept>

using namespace std;

// An exception class for RangeArray.
```

```

class RAExc {
    string err;
public:

    RAExc(string e) {
        err = e;
    }

    string geterr() { return err; }
};

// A range-selectable array container.
template<class T, class Allocator = allocator<T> >
class RangeArray {
    T *arrayptr; // pointer to array that underlies the container

    unsigned len; // holds length of the container
    int upperbound; // lower bound
    int lowerbound; // upper bound
    Allocator a; // allocator
public:

    // Required typedefs for container.
    typedef T value_type;
    typedef Allocator allocator_type;
    typedef typename Allocator::reference reference;
    typedef typename Allocator::const_reference const_reference;
    typedef typename Allocator::size_type size_type;
    typedef typename Allocator::difference_type difference_type;
    typedef typename Allocator::pointer pointer;
    typedef typename Allocator::const_pointer const_pointer;

    // Forward iterators.
    typedef T * iterator;
    typedef const T * const_iterator;

    // Note: This container does not support reverse
    // iterators, but you can add them if you like.

    // ***** Constructors and Destructor *****

    // Default constructor.
    RangeArray()
    {
        upperbound = lowerbound = 0;
        len = 0;
        arrayptr = a.allocate(0);
    }

    // Construct an array of the specified range
    // with each element having the specified initial value.
    RangeArray(int low, int high, const T &t);

```

```

// Construct zero-based array of num elements
// with the value t. This constructor is required
// for STL compatibility.
RangeArray(int num, const T &t=T());

// Construct from range of iterators.
RangeArray(iterator start, iterator stop);

// Copy constructor.
RangeArray(const RangeArray &o);

// Destructor.
~RangeArray();

// ***** Operator Functions *****

// Return reference to specified element.
T &operator[](int i)
{
    return arrayptr[i - lowerbound];
}

// Return const references to specified element.
const T &operator[](int i) const
{
    return arrayptr[i - lowerbound];
}

// Assign one container to another.
RangeArray &operator=(const RangeArray &o);

// ***** Insert Functions *****

// Insert val at p.
iterator insert(iterator p, const T &val);

// Insert num copies of val at p.
void insert(iterator p, int num, const T &val)
{
    for(; num>0; num--) p = insert(p, val) + 1;
}

// Insert range specified by start and stop at p.
void insert(iterator p, iterator start, iterator stop)
{
    while(start != stop) {
        p = insert(p, *start) + 1;
        start++;
    }
}

```

```

}

// ***** Erase Functions *****

// Erase element at p.
iterator erase(iterator p);

// Erase specified range.
iterator erase(iterator start, iterator stop)
{
    iterator p = end();

    for(int i=stop-start; i > 0; i--)
        p = erase(start);

    return p;
}

// ***** Push and Pop Functions *****

// Add element to end.
void push_back(const T &val)
{
    insert(end(), val);
}

// Remove element from end.
void pop_back()
{
    erase(end()-1);
}

// Add element to front.
void push_front(const T &val)
{
    insert(begin(), val);
}

// Remove element from front.
void pop_front()
{
    erase(begin());
}

// ***** front() and back() functions *****

// Return reference to first element.
T &front()
{

```

```

    return arrayptr[0];
}

// Return const reference to first element.
const T &front() const
{
    return arrayptr[0];
}

// Return reference to last element.
T &back()
{
    return arrayptr[len-1];
}

// Return const reference to last element.
const T &back() const
{
    return arrayptr[len-1];
}

// ***** Iterator Functions *****

// Return iterator to first element.
iterator begin()
{
    return &arrayptr[0];
}

// Return iterator to last element.
iterator end()
{
    return &arrayptr[upperbound - lowerbound];
}

// Return const iterator to first element.
const_iterator begin() const
{
    return &arrayptr[0];
}

// Return const iterator to last element.
const_iterator end() const
{
    return &arrayptr[upperbound - lowerbound];
}

// ***** Misc. Functions *****

// The at() function performs a range check.

```

```

// Return a reference to the specified element.
T &at(int i)
{
    if(i < lowerbound || i >= upperbound)
        throw out_of_range("Index Out of Range");

    return arrayptr[i - lowerbound];
}

// Return a const reference to the specified element.
const T &at(int i) const
{
    if(i < lowerbound || i >= upperbound)
        throw out_of_range("Index Out of Range");

    return arrayptr[i - lowerbound];
}

// Return the size of the container.
size_type size() const
{
    return end() - begin();
}

// Return the maximum size of a RangeArray.
size_type max_size()
{
    return a.max_size();
}

// Return true if container is empty.
bool empty()
{
    return size() == 0;
}

// Exchange the values of two containers.
void swap(RangeArray &b)
{
    RangeArray<T> tmp;

    tmp = *this;
    *this = b;
    b = tmp;
}

// Remove and destroy all elements.
void clear()
{
    erase(begin(), end());
}

```

```

// ***** Non-STL functions *****

// Return endpoints.
int getlowerbound()
{
    return lowerbound;
}

int getupperbound()
{
    return upperbound;
}

};

// ***** Implementations of non-inline functions *****

// Construct an array of the specified range
// with each element having the specified initial value.
template <class T, class A>
RangeArray<T, A>::RangeArray(int low, int high,
                             const T &t)
{
    if(high <= low) throw RAExc("Invalid Range");

    high++;

    // Save endpoints.
    upperbound = high;
    lowerbound = low;

    // Allocate memory for the container.
    arrayptr = a.allocate(high - low);

    // Save the length of the container.
    len = high - low;

    // Construct the elements.
    for(size_type i=0; i < size(); i++)
        a.construct(&arrayptr[i], t);
}

// Construct zero-based array of num elements
// with the value t. This constructor is required
// for STL compatibility.
template <class T, class A>
RangeArray<T, A>::RangeArray(int num, const T &t) {

    // Save endpoints.
    upperbound = num;
    lowerbound = 0;
}

```



```

    // Allocate memory for the container.
    arrayptr = a.allocate(num);

    // Save the length of the container.
    len = num;

    // Construct the elements.
    for(size_type i=0; i < size(); i++)
        a.construct(&arrayptr[i], t);
}

// Construct zero-based array from range of iterators.
// This constructor is required for STL compatibility.
template <class T, class A>
RangeArray<T, A>::RangeArray(iterator start,
                             iterator stop)
{
    // Allocate sufficient memory.
    arrayptr = a.allocate(stop - start);

    upperbound = stop - start;
    lowerbound = 0;

    len = stop - start;

    // Construct the elements using those
    // specified by the range of iterators.
    for(size_type i=0; i < size(); i++)
        a.construct(&arrayptr[i], *start++);
}

// Copy constructor.
template <class T, class A>
RangeArray<T, A>::RangeArray(const RangeArray<T, A> &o)
{
    // Allocate memory for the copy.
    arrayptr = a.allocate(o.size());

    upperbound = o.upperbound;
    lowerbound = o.lowerbound;
    len = o.len;

    // Make the copy.
    for(size_type i=0; i < size(); i++)
        a.construct(&arrayptr[i], o.arrayptr[i]);
}

// Destructor.
template <class T, class A>
RangeArray<T, A>::~RangeArray()
{

```

```

// Call destructors for elements in the container.
for(size_type i=0; i < size(); i++)
    a.destroy(&arrayptr[i]);

// Release memory.
a.deallocate(arrayptr, size());
}

// Assign one container to another.
template <class T, class A> RangeArray<T, A> &
RangeArray<T, A>::operator=(const RangeArray<T, A> &o)
{
    // Call destructors for elements in target container.
    for(size_type i=0; i < size(); i++)
        a.destroy(&arrayptr[i]);

    // Release original memory.
    a.deallocate(arrayptr, size());

    // Allocate memory for new size.
    arrayptr = a.allocate(o.size());

    upperbound = o.upperbound;
    lowerbound = o.lowerbound;
    len = o.len;

    // Make copy.
    for(size_type i=0; i < size(); i++)
        arrayptr[i] = o.arrayptr[i];

    return *this;
}

// Insert val at p.
template <class T, class A>
typename RangeArray<T, A>::iterator
RangeArray<T, A>::insert(iterator p, const T &val)
{
    iterator q;
    size_type i, j;

    // Get sufficient memory.
    T *tmp = a.allocate(size() + 1);

    // Copy existing elements to new array,
    // inserting new element if possible.
    for(i=j=0; i < size(); i++, j++) {
        if(&arrayptr[i] == p) {
            tmp[j] = val;
            q = &tmp[j];
            j++;
        }
    }

```

```

    tmp[j] = arrayptr[i];
}

// Otherwise, the new element goes on end.
if(p == end()) {
    tmp[j] = val;
    q = &tmp[j];
}

// Adjust len and bounds.
len++;
if(p < &arrayptr[abs(lowerbound)])
    lowerbound--;
else
    upperbound++;

// Call destructors for elements in old container.
for(size_type i=0; i < size()-1; i++)
    a.destroy(&arrayptr[i]);

// Release memory for old container.
a.deallocate(arrayptr, size()-1);

arrayptr = tmp;

return q;
}

// Erase element at p.
template <class T, class A>
typename RangeArray<T, A>::iterator
RangeArray<T, A>::erase(iterator p)
{
    iterator q = p;
    ..

    // Destruct element being erased.
    if(p != end()) a.destroy(p);

    // Adjust len and bounds.
    len--;
    if(p < &arrayptr[abs(lowerbound)])
        lowerbound++;
    else
        upperbound--;

    // Compact remaining elements.
    for( ; p < end(); p++)
        *p = *(p+1);

    return q;
}

```

```
// ***** Relational Operators *****

template<class T, class Allocator>
    bool operator==(const RangeArray<T, Allocator> &a,
                    const RangeArray<T, Allocator> &b)
{
    if(a.size() != b.size()) return false;

    return equal(a.begin(), a.end(), b.begin());
}

template<class T, class Allocator>
    bool operator!=(const RangeArray<T, Allocator> &a,
                    const RangeArray<T, Allocator> &b)
{
    if(a.size() != b.size()) return true;

    return !equal(a.begin(), a.end(), b.begin());
}

template<class T, class Allocator>
    bool operator<(const RangeArray<T, Allocator> &a,
                  const RangeArray<T, Allocator> &b)
{
    return lexicographical_compare(a.begin(), a.end(),
                                   b.begin(), b.end());
}

template<class T, class Allocator>
    bool operator>(const RangeArray<T, Allocator> &a,
                  const RangeArray<T, Allocator> &b)
{
    return b < a;
}

template<class T, class Allocator>
    bool operator<=(const RangeArray<T, Allocator> &a,
                   const RangeArray<T, Allocator> &b)
{
    return !(a > b);
}

template<class T, class Allocator>
    bool operator>=(const RangeArray<T, Allocator> &a,
                   const RangeArray<T, Allocator> &b)
{
    return !(a < b);
}
```

正如代码开始的注释所指出的那样，您应该将所有的代码都放到一个文件 `ra.h` 中。在后面所示的示例程序中将会使用它。

前面的代码包含两个类。第一个是 `RAExc` 异常类。如果使用无效的边界创建一个 `RangeArray`，如下界大于上界，则会抛出这种类型的异常。第二个是 `RangeArray` 容器类，在下面部分将详细介绍这个类。

8.4.3 详细讨论 `RangeArray` 类

如同所有内建的 STL 序列式容器一样，`RangeArray` 以如下的模板说明开始：

```
template<class T, class Allocator=allocator<T>>
```

`T` 是容器中存储的数据类型，`Allocator` 是分配器，默认情况下是标准分配器。

1. 私有成员

`RangeArray` 数组类以如下的私有声明开始：

```
T *arrayptr; // pointer to array that underlies the container

unsigned len; // holds length of the container
int upperbound; // lower bound
int lowerbound; // upper bound

Allocator a; // allocator
```

指针 `arrayptr` 存储了一个指针，这个指针指向的内存将包含元素类型为 `T` 的数组。这块内存将存储类型为 `RangeArray` 的对象所保存的元素。这个数组的索引通常从 0 开始。`RangeArray` 对象的索引将从基于 0 的索引转换为 `arrayptr` 所指向的数组。

`RangeArray` 当前的长度存储在 `len` 中。数组的上界和下界分别存储在 `upperbound` 和 `lowerbound` 中。对于 `RangeArray` 来说，0 被认为是正数。容器的分配器存储在 `a` 中。

2. 所需的类型定义

在私有成员之后，`RangeArray` 定义了所有序列式容器需要的各种 `typedef`，如下所示：

```
// Required typedefs for container.
typedef T value_type;
typedef Allocator allocator_type;
typedef typename Allocator::reference reference;
typedef typename Allocator::const_reference const_reference;
typedef typename Allocator::size_type size_type;
typedef typename Allocator::difference_type difference_type;
typedef typename Allocator::pointer pointer;
typedef typename Allocator::const_pointer const_pointer;

// Forward iterators.
typedef T * iterator;
typedef const T * const_iterator;
```

这些 `typedef` 类似于内建的 STL 容器使用的 `typedef`。注意，前向迭代器只是指向类型 `T` 的对象的一些指针。对于 `RangeArray` 而言，这就足够了，但是对于更加复杂的容器，可能并非如此。另外，在此没有提供反向迭代器，但是您可以将其作为一个有趣的练习试着加入它们。

3. RangeArray 的构造函数和析构函数

为了创建一个兼容的容器，您必须支持前面描述的为序列式容器定义的 4 个构造函数。RangeArray 还定义了第 5 个构造函数，允许您创建一个指定范围的数组。这些构造函数如下所示：

```
// Default constructor.
RangeArray()
{
    upperbound = lowerbound = 0;
    len = 0;
    arrayptr = a.allocate(0);
}

// Construct an array of the specified range
// with each element having the specified initial value.
template <class T, class A>
RangeArray<T, A>::RangeArray(int low, int high,
                             const T &t)
{
    if (high <= low) throw RAExc("Invalid Range");

    high++;

    // Save endpoints.
    upperbound = high;
    lowerbound = low;

    // Allocate memory for the container.
    arrayptr = a.allocate(high - low);

    // Save the length of the container.
    len = high - low;

    // Construct the elements.
    for (size_type i=0; i < size(); i++)
        a.construct(&arrayptr[i], t);
}

// Construct zero-based array of num elements
// with the value t. This constructor is required
// for STL compatibility.
template <class T, class A>
RangeArray<T, A>::RangeArray(int num, const T &t) {

    // Save endpoints.
    upperbound = num;
    lowerbound = 0;

    // Allocate memory for the container.
    arrayptr = a.allocate(num);
```

```

// Save the length of the container.
len = num;

// Construct the elements.
for(size_type i=0; i < size(); i++)
    a.construct(&arrayptr[i], t);
}

// Construct zero-based array from range of iterators.
// This constructor is required for STL compatibility.
template <class T, class A>
RangeArray<T, A>::RangeArray(iterator start,
                             iterator stop)
{
    // Allocate sufficient memory.
    arrayptr = a.allocate(stop - start);

    upperbound = stop - start;
    lowerbound = 0;

    len = stop - start;

    // Construct the elements using those
    // specified by the range of iterators.
    for(size_type i=0; i < size(); i++)
        a.construct(&arrayptr[i], *start++);
}

// Copy constructor.
template <class T, class A>
RangeArray<T, A>::RangeArray(const RangeArray<T, A> &o)
{
    // Allocate memory for the copy.
    arrayptr = a.allocate(o.size());

    upperbound = o.upperbound;
    lowerbound = o.lowerbound;
    len = o.len;

    // Make the copy.
    for(size_type i=0; i < size(); i++)
        a.construct(&arrayptr[i], o.arrayptr[i]);
}

```

第一个构造函数是默认构造函数，它创建一个空的对象。STL 指定的限制之一是对默认对象调用 `size()` 的结果必须为 0。因此，默认构造函数将上界和下界设置为 0。另外，它还将 `len` 设置为 0，并且使用分配器提供的 `allocate()` 函数创建了一个长度为 0 的数组。最后两步确保在所有情况下都存在完整的对象。

第二个构造函数是 `RangeArray` 特有的。它创建了一个指定范围的对象，每个元素都具有一

定的初始值。下界在第一个参数中指定；上界在第二个参数中指定。如果上界小于或者等于下界，就会抛出 `RAExc` 异常。第三个参数传递了初始值。因此，这条语句：

```
RangeArray<char> ch(-2,10, 'X');
```

创建了一个字符数组，从 -2 开始一直到 10，每个元素的初始值都是 X。这个数组使用 `allocate()` 来分配，`allocate()` 是 `allocator` 类的一个成员函数。为了创建一个兼容的容器，您必须使用分配器函数而不是 `new` 函数来获取容器需要的内存。一旦完成分配，数组中的每个元素都使用作为第三个参数传递的值来创建。(如果可以使用默认的初始化值，可能会更方便，因为不需要指定这个值。但是这样做会与第三个构造函数混淆，STL 需要第三个构造函数)。这个创建过程是通过使用另一个分配器函数 `construct()` 完成的。

第三个构造函数创建了一个基数为 0 的数组，这个数组具有指定数量的元素，使用指定的值初始化每个元素。可以使用默认的初始化值。这个构造函数不是只对 `RangeArray` 有用，而且是被 STL 容器规范要求的。

第四个构造函数创建了一个给定值范围的、基数为 0 的数组。这个范围是由传递的指向范围的开头以及结尾的迭代器指定的。这个构造函数不仅对于 `RangeArray` 是有用的，而且是被 STL 容器规范所要求的。

最后一个是复制构造函数。它为一个新的对象分配内存，并从源对象复制边界、长度以及元素。因此，这个副本具有自己的内存，但是其他方面与源对象相同。

`RangeArray` 的析构函数如下所示：

```
// Destructor.
template <class T, class A>
RangeArray<T, A>::~~RangeArray()
{
    // Call destructors for elements in the container.
    for(size_type i=0; i < size(); i++)
        a.destroy(&arrayptr[i]);

    // Release memory.
    a.deallocate(arrayptr, size());
}
```

首先，它通过调用分配器的 `destroy()` 函数销毁数组中的每个元素。然后通过调用分配器的 `deallocate()` 函数释放数组使用的内存。

记住，`RangeArray` 的构造函数没有使用 `new` 分配内存，析构函数也不会使用 `delete` 释放内存。相反，它们使用了分配器提供的合适的成员函数。这种方式使得用户可以指定管理容器内存的另一种方法。

4. `RangeArray` 的运算符函数

`RangeArray` 定义了 3 个成员运算符函数。前两个 `operator[]()` 函数如下所示：

```
// Return reference to specified element.
T &operator[](int i)
{
    return arrayptr[i - lowerbound];
}
```



```

}

// Return const references to specified element.
const T &operator[](int i) const
{
    return arrayptr[i - lowerbound];
}

```

对于完整的容器，需要[]运算符的 const 和非 const 版本。这些函数提供了检索 RangeArray 的机制。特别注意检索 arrayptr 的方式。记住，arrayptr 指向一个标准的 C++ 数组。因此，i 中传递的索引必须转换为 arrayptr 中基数为 0 的索引。lowerbound 保存了对应于 RangeArray 中最低索引的值。因此，为了获取基数为 0 的索引，从 i 中减去 lowerbound。

另一个需要注意的问题：operator[]() 没有执行边界检查。STL 没有要求这个运算符执行边界检查，在此也没有包括。这是合理的，因为除了使得用户指定的范围之外，RangeArray 的行为类似于普通的数组。（记住，普通的 C++ 数组没有提供边界检查）。当然，如果您愿意，可以在 operator[]() 中加入边界检查。

下面所示的 operator=() 函数有一点复杂：

```

// Assign one container to another.
template <class T, class A> RangeArray<T, A> &
RangeArray<T, A>::operator=(const RangeArray<T, A> &o)
{
    // Call destructors for elements in target container.
    for(size_type i=0; i < size(); i++)
        a.destroy(&arrayptr[i]);

    // Release original memory.
    a.deallocate(arrayptr, size());

    // Allocate memory for new size.
    arrayptr = a.allocate(o.size());

    upperbound = o.upperbound;
    lowerbound = o.lowerbound;
    len = o.len;

    // Make copy.
    for(size_type i=0; i < size(); i++)
        arrayptr[i] = o.arrayptr[i];

    return *this;
}

```

在这个程序中，首先销毁并释放目标对象中存在的任何对象。然后分配足够的内存来保存源对象中的内容。随后恰当地设置成员变量，然后复制元素。最后返回一个目标对象的引用。

5. insert() 函数

STL 要求序列式容器支持三种形式的 insert()：一个用来插入值，一个用来插入一个值的多

个副本，还有一个用来确定范围。第一个版本的 insert() 如下所示：

```
// Insert val at p.
template <class T, class A>
typename RangeArray<T, A>::iterator
RangeArray<T, A>::insert(iterator p, const T &val)
{
    iterator q;
    size_type i, j;

    // Get sufficient memory.
    T *tmp = a.allocate(size() + 1);

    // Copy existing elements to new array,
    // inserting new element if possible.
    for(i=j=0; i < size(); i++, j++) {
        if(&arrayptr[i] == p) {
            tmp[j] = val;
            q = &tmp[j];
            j++;
        }
        tmp[j] = arrayptr[i];
    }

    // Otherwise, the new element goes on end.
    if(p == end()) {
        tmp[j] = val;
        q = &tmp[j];
    }

    // Adjust len and bounds.
    len++;
    if(p < &arrayptr[abs(lowerbound)])
        lowerbound--;
    else
        upperbound++;

    // Call destructors for elements in old container.
    for(size_type i=0; i < size()-1; i++)
        a.destroy(&arrayptr[i]);

    // Release memory for old container.
    a.deallocate(arrayptr, size()-1);

    arrayptr = tmp;

    return q;
}
```

第一个版本以一个迭代器作为第一个参数，在迭代器指定的位置插入一个元素。这个函数返回一个插入元素的迭代器。为了保存现有的元素和新加入的元素，它分配了一个足够大的内

存片断。然后将现有的元素复制到新分配的内存中，并将新的元素插入到合适的位置。随后恰当地更新了 `len` 以及 `upperbound`(或者 `lowerbound`)。在下一步，它从原来的 `RangeArray` 销毁并释放了对象，并将新内存的地址赋给 `arrayptr`。最后，返回一个指向插入对象的指针。

现在，仔细观察更新 `lowerbound` 或者 `upperbound` 变量的代码。数组可以向正方向或者负方向增长，这取决于新的元素是在数组的正方向还是负方向插入。因此，有必要判断插入发生的位置，并改变合适的值。这个判断是通过比较 `p` 中传递的迭代器以及 `arrayptr[lowerbound]` 中元素的地址而做出的。如果迭代器小于这个元素，负方向扩展；否则，正方向扩展。

第二个版本的 `insert()` 如下所示：

```
// Insert num copies of val at p.
void insert(iterator p, int num, const T &val)
{
    for(; num>0; num--) p = insert(p, val) + 1;
}

// Insert range specified by start and stop at p.
void insert(iterator p, iterator start, iterator stop)
{
    while(start != stop) {
        p = insert(p, *start) + 1;
        start++;
    }
}
```

6. erase 函数

序列式容器必须支持两种版本的 `erase()` 函数。第一种版本的 `erase()` 函数删除了迭代器所指的元素。当删除了某个元素之后，立刻返回指向这个元素的迭代器，如果最后一个元素被删除，则返回 `end()`。这个版本的 `erase()` 函数如下所示：

```
// Erase element at p.
template <class T, class A>
typename RangeArray<T, A>::iterator
RangeArray<T, A>::erase(iterator p)
{
    iterator q = p;

    // Destruct element being erased.
    if(p != end()) a.destroy(p);

    // Adjust len and bounds.
    len--;
    if(p < &arrayptr[abs(lowerbound)])
        lowerbound++;
    else
        upperbound--;

    // Compact remaining elements.
    for( ; p < end(); p++)
        *p = *(p+1);
}
```

```
    return q;
}
```

这个版本的 `erase()` 函数通过销毁被删除的元素、合适地调整上下界，然后压缩剩余的元素来完成操作。

第二个版本采用了第一个版本的框架，如下所示。它删除了多个元素。它返回指向这个范围中最后元素的迭代器(也就是说，`stop` 指向的那个元素)。因此，它删除了在 `start` 和 `stop - 1` 之间的元素。

```
// Erase specified range.
iterator erase(iterator start, iterator stop)
{
    iterator p = end();

    for(int i=stop-start; i > 0; i--)
        p = erase(start);

    return p;
}
```

7. 压入与弹出函数

`RangeArray` 实现了 `push_back()`、`pop_back()`、`push_front()` 以及 `pop_front()`，如下所示。正如您所看到的那样，它们是通过 `insert()` 和 `erase()` 实现的，并且它们的操作方式相当直接：

```
// Add element to end.
void push_back(const T &val)
{
    insert(end(), val);
}

// Remove element from end.
void pop_back()
{
    erase(end()-1);
}

// Add element to front.
void push_front(const T &val)
{
    insert(begin(), val);
}

// Remove element from front.
void pop_front()
{
    erase(begin());
}
```

8. front()和back()函数

front()和back()函数各自简单地返回了指向数组开始和结尾的迭代器，如下所示。它们的实现相当直接。然而要注意，const 和非 const 的两个版本都需要。

```
// Return reference to first element.
T &front()
{
    return arrayptr[0];
}

// Return const reference to first element.
const T &front() const
{
    return arrayptr[0];
}

// Return reference to last element.
T &back()
{
    return arrayptr[len-1];
}

// Return const reference to last element.
const T &back() const
{
    return arrayptr[len-1];
}
```

9. 迭代器函数

由于 RangeArray 容器的迭代器只是简单的指针，保存在 arrayptr 指向的内存中。迭代器函数 begin()和end()很小。注意 const 和非 const 的两个版本都需要。

```
// Return iterator to first element.
iterator begin()
{
    return &arrayptr[0];
}

// Return iterator to last element.
iterator end()
{
    return &arrayptr[upperbound - lowerbound];
}

// Return const iterator to first element.
const_iterator begin() const
{
    return &arrayptr[0];
}
```

```
// Return const iterator to last element.
const_iterator end() const
{
    return &arrayptr[upperbound - lowerbound];
}
```

10. 其他函数

所有的序列式容器都必须实现 `size()`、`max_size()`、`empty()`、`swap()` 以及 `clear()`。这些函数如下所示：

```
// Return the size of the container.
size_type size() const
{
    return end() - begin();
}

// Return the maximum size of a RangeArray.
size_type max_size()
{
    return a.max_size();
}

// Return true if container is empty.
bool empty()
{
    return size() == 0;
}

// Exchange the values of two containers.
void swap(RangeArray &b)
{
    RangeArray<T> tmp;
    tmp = *this;
    *this = b;
    b = tmp;
}

// Remove and destroy all elements.
void clear()
{
    erase(begin(), end());
}
```

通常，凭直觉就可以操作这些函数。然而，`max_size()` 值得花一些文字来描述。它返回能够创建的最大容器所能包含的 T 类型元素的数量。这个值通过调用分配器定义的 `max_size()` 函数获得。

`RangeArray` 还实现了可选的 `at()` 函数，如下所示：

```
// The at() function performs a range check.
// Return a reference to the specified element.
```

```

T &at(int i)
{
    if(i < lowerbound || i >= upperbound)
        throw out_of_range("Index Out of Range");

    return arrayptr[i - lowerbound];
}

// Return a const reference to the specified element.
const T &at(int i) const
{
    if(i < lowerbound || i >= upperbound)
        throw out_of_range("Index Out of Range");

    return arrayptr[i - lowerbound];
}

```

at()函数返回指定索引的位置处元素的引用。它与 operator[]()仅有的区别是它对索引的范围执行检查。如果这个索引越界, at()就会抛出 out_of_range 异常。这个异常在 C++的头文件 <stdexcept>中定义。

RangeArray 还提供了非 STL 函数 getlowerbound()和 getupperbound()。这两个函数分别获取 RangeArray 的上界和下界, 如下所示:

```

// Return endpoints.
int getlowerbound()
{
    return lowerbound;
}

int getupperbound()
{
    return upperbound;
}

```

11. 关系运算符

为 RangeArray 定义的关系运算符如下所示:

```

template<class T, class Allocator>
bool operator==(const RangeArray<T, Allocator> &a,
                const RangeArray<T, Allocator> &b)
{
    if(a.size() != b.size()) return false;

    return equal(a.begin(), a.end(), b.begin());
}

template<class T, class Allocator>
bool operator!=(const RangeArray<T, Allocator> &a,
                const RangeArray<T, Allocator> &b)
{

```

```

    if(a.size() != b.size()) return true;

    return !equal(a.begin(), a.end(), b.begin());
}

template<class T, class Allocator>
    bool operator<(const RangeArray<T, Allocator> &a,
                   const RangeArray<T, Allocator> &b)
{
    return lexicographical_compare(a.begin(), a.end(),
                                    b.begin(), b.end());
}

template<class T, class Allocator>
    bool operator>(const RangeArray<T, Allocator> &a,
                   const RangeArray<T, Allocator> &b)
{
    return b < a;
}

template<class T, class Allocator>
    bool operator<=(const RangeArray<T, Allocator> &a,
                    const RangeArray<T, Allocator> &b)
{
    return !(a > b);
}

template<class T, class Allocator>
    bool operator>=(const RangeArray<T, Allocator> &a,
                    const RangeArray<T, Allocator> &b)
{
    return !(a < b);
}

```

`operator=()`和 `operator!=()`都使用了 `equal()`算法来判断等同性。如同 `equal()`定义的那样，如果两个对象以相同的顺序包含了相同的元素，这两个对象就是相等的。

运算符“<”使用了 `lexicographical_compare()`来判断一个对象何时小于另一个对象。标准 C++推荐使用这个函数。它通过比较两个序列中对应的元素，并查找第一个不相等的元素。如果能找到，当第一个范围的元素比第二个范围的元素小时，返回 `true`。否则返回 `false`。

8.4.4 一些 RangeArray 示例程序

为了演示 `RangeArray`，下面给出了 3 个示例程序。第一个演示不同的成员函数，如下所示。这个程序还使用了 3 个算法，一个函数对象以及一个绑定器。这些元素的使用说明了 `RangeArray` 是功能齐全的、与 STL 其余部分兼容的容器。

```

// Demonstrate basic RangeArray operations.
#include <iostream>
#include <algorithm>
#include <functional>

```



```

#include "ra.h"
using namespace std;

// Display integers -- for use by for_each.
void display(int v)
{
    cout << v << " ";
}

int main()
{
    RangeArray<int> ob(-5, 5, 0);
    RangeArray<int>::iterator p;
    int i, sum;
    cout << "Size of ob is: " << ob.size() << endl;

    cout << "Initial contents of ob:\n";
    for(i=-5; i <= 5; i++) cout << ob[i] << " ";
    cout << endl;

    // Give ob some values.
    for(i=-5; i <= 5; i++) ob[i] = i;

    cout << "New values for ob: \n";
    p = ob.begin();
    do {
        cout << *p++ << " ";
    } while (p != ob.end());
    cout << endl;

    // Display sum of negative indexes.
    sum = 0;
    for(i = ob.getlowerbound(); i < 0; i++)
        sum += ob[i];
    cout << "Sum of values with negative subscripts is: ";
    cout << sum << "\n\n";

    // Use copy() algorithm to copy one object to another.
    cout << "Copy ob to ob2 using copy() algorithm.\n";

    RangeArray<int> ob2(-5, 5, 0);
    copy(ob.begin(), ob.end(), ob2.begin());

    // Use for_each() algorithm to display ob2.
    cout << "Contents of ob2: \n";
    for_each(ob2.begin(), ob2.end(), display);
    cout << "\n\n";

    // Use replace_copy_if() algorithm to remove values less than zero.
    cout << "Replace values less than zero with zero.\n";
    cout << "Put the result into ob3.\n";
    RangeArray<int> ob3(ob.begin(), ob.end());

```

```

// The next line uses the function object less() and
// the binder bind2nd().
replace_copy_if(ob.begin(), ob.end(), ob3.begin(),
               bind2nd(less<int>(), 0), 0);
cout << "Contents of ob3: \n";
for_each(ob3.begin(), ob3.end(), display);
cout << "\n\n";

cout << "Swap ob and ob3.\n";
ob.swap(ob3); // swap ob and ob3
cout << "Here is ob3:\n";
for_each(ob3.begin(), ob3.end(), display);
cout << endl;
cout << "Swap again to restore.\n";
ob.swap(ob3); // restore
cout << "Here is ob3 after second swap:\n";
for_each(ob3.begin(), ob3.end(), display);
cout << "\n\n";

// Use insert() member functions.
cout << "Element at ob[0] is " << ob[0] << endl;
cout << "Insert values into ob.\n";
ob.insert(ob.end(), -9999);
ob.insert(&ob[1], 99);
ob.insert(&ob[-3], -99);
for_each(ob.begin(), ob.end(), display);
cout << endl;
cout << "Element at ob[0] is " << ob[0] << "\n\n";

cout << "Insert -7 three times to front of ob.\n";
ob.insert(ob.begin(), 3, -7);
for_each(ob.begin(), ob.end(), display);
cout << endl;
cout << "Element at ob[0] is " << ob[0] << "\n\n";

// Use push_back() and pop_back().
cout << "Push back the value 40 onto ob.\n";
ob.push_back(40);
for_each(ob.begin(), ob.end(), display);
cout << endl;
cout << "Pop back two values from ob.\n";
ob.pop_back(); ob.pop_back();
for_each(ob.begin(), ob.end(), display);
cout << "\n\n";

// Use push_front() and pop_front().
cout << "Push front the value 19 onto ob.\n";
ob.push_front(19);
for_each(ob.begin(), ob.end(), display);
cout << endl;
cout << "Pop front two values from ob.\n";

```

```

ob.pop_front(); ob.pop_front();
for_each(ob.begin(), ob.end(), display);
cout << "\n\n";

// Use front() and back()
cout << "ob.front(): " << ob.front() << endl;
cout << "ob.back(): " << ob.back() << "\n\n";

// Use erase().
cout << "Erase element at 0.\n";
p = ob.erase(&ob[0]);
for_each(ob.begin(), ob.end(), display);
cout << endl;
cout << "Element at ob[0] is " << ob[0] << endl;
cout << endl;

cout << "Erase many elements in ob.\n";
p = ob.erase(&ob[-2], &ob[3]);
for_each(ob.begin(), ob.end(), display);
cout << endl;
cout << "Element at ob[0] is " << ob[0] << endl;
cout << endl;

cout << "Insert ob4 into ob.\n";
RangeArray<int> ob4(0, 2, 0);
for(i=0; i < 3; i++) ob4[i] = i+100;
ob.insert(&ob[0], ob4.begin(), ob4.end());
for_each(ob.begin(), ob.end(), display);
cout << endl;
cout << "Element at ob[0] is " << ob[0] << endl;
cout << endl;

cout << "Here is ob shown with its indices:\n";
for(i=ob.getlowerbound(); i<ob.getupperbound(); i++)
cout << "[" << i << "]: " << ob[i] << endl;
cout << endl;

// Use the at() function.
cout << "Use the at() function.\n";
for(i=ob.getlowerbound(); i < ob.getupperbound(); i++)
    ob.at(i) = i * 11;

for(i=ob.getlowerbound(); i < ob.getupperbound(); i++)
    cout << ob.at(i) << " ";
cout << "\n\n";

// Use the clear() function.
cout << "Clear ob.\n";
ob.clear();
for_each(ob.begin(), ob.end(), display); // no effect!
cout << "Size of ob after clear: " << ob.size()
    << "\nBounds: " << ob.getlowerbound()

```

```

    << " to " << ob.getupperbound() << "\n\n";

    // Create a copy of an object.
    cout << "Make a copy of ob2.\n";
    RangeArray<int> ob5(ob2);
    for_each(ob5.begin(), ob5.end(), display);
    cout << "\n\n";

    // Construct a new object from a range.
    cout << "Construct object from a range.\n";
    RangeArray<int> ob6(&ob2[-2], ob2.end());
    cout << "Size of ob6: " << ob6.size() << endl;
    for_each(ob6.begin(), ob6.end(), display);
    cout << endl;

    return 0;
}

```

这个程序的输出如下所示:

```

Size of ob is: 11
Initial contents of ob:
0 0 0 0 0 0 0 0 0 0 0
New values for ob:
-5 -4 -3 -2 -1 0 1 2 3 4 5
Sum of values with negative subscripts is: -15

Copy ob to ob2 using copy() algorithm.
Contents of ob2:
-5 -4 -3 -2 -1 0 1 2 3 4 5

Replace values less than zero with zero.
Put the result into ob3.
Contents of ob3:
0 0 0 0 0 0 1 2 3 4 5

Swap ob and ob3.
Here is ob3:
-5 -4 -3 -2 -1 0 1 2 3 4 5
Swap again to restore.

Here is ob3 after second swap:
0 0 0 0 0 0 1 2 3 4 5

Element at ob[0] is 0
Insert values into ob.
-5 -4 -99 -3 -2 -1 0 99 1 2 3 4 5 -9999
Element at ob[0] is 0

Insert -7 three times to front of ob.
-7 -7 -7 -5 -4 -99 -3 -2 -1 0 99 1 2 3 4 5 -9999
Element at ob[0] is 0

```

Push back the value 40 onto ob.

```
-7 -7 -7 -5 -4 -99 -3 -2 -1 0 99 1 2 3 4 5 -9999 40
```

Pop back two values from ob.

```
-7 -7 -7 -5 -4 -99 -3 -2 -1 0 99 1 2 3 4 5
```

Push front the value 19 onto ob.

```
19 -7 -7 -7 -5 -4 -99 -3 -2 -1 0 99 1 2 3 4 5
```

Pop front two values from ob.

```
-7 -7 -5 -4 -99 -3 -2 -1 0 99 1 2 3 4 5
```

```
ob.front(): -7
```

```
ob.back(): 5
```

Erase element at 0.

```
-7 -7 -5 -4 -99 -3 -2 -1 99 1 2 3 4 5
```

Element at ob[0] is 99

Erase many elements in ob.

```
-7 -7 -5 -4 -99 -3 3 4 5
```

Element at ob[0] is 3

Insert ob4 into ob.

```
-7 -7 -5 -4 -99 -3 100 101 102 3 4 5
```

Element at ob[0] is 100

Here is ob shown with its indices:

```
[-6]: -7
```

```
[-5]: -7
```

```
[-4]: -5
```

```
[-3]: -4
```

```
[-2]: -99
```

```
[-1]: -3
```

```
[0]: 100
```

```
[1]: 101
```

```
[2]: 102
```

```
[3]: 3
```

```
[4]: 4
```

```
[5]: 5
```

Use the at() function.

```
-66 -55 -44 -33 -22 -11 0 11 22 33 44 55
```

Clear ob.

Size of ob after clear: 0

Bounds: 0 to 0

Make a copy of ob2.

```
-5 -4 -3 -2 -1 0 1 2 3 4 5
```

Construct object from a range.

Size of ob6: 8

```
-2 -1 0 1 2 3 4 5
```

下一个示例程序演示了关系运算符:

```
// Demonstrate the relational operators.
#include <iostream>
#include "ra.h"

using namespace std;

// Display integers -- for use by for_each.
void display(int v)
{
    cout << v << " ";
}

int main()
{
    RangeArray<int> ob1(-3, 2, 0), ob2(-3, 2, 0), ob3(-4, 4, 0);
    int i;

    // Give ob1 and ob2 some values.
    for(i = -3; i < 3; i++) {
        ob1[i] = i;
        ob2[i] = i;
    }

    cout << "Contents of ob1 and ob2:\n";
    for(i=-3; i < 3; i++)
        cout << ob1[i] << " ";
    cout << endl;

    for(i=-3; i < 3; i++)
        cout << ob2[i] << " ";
    cout << "\n\n";

    if(ob1 == ob2) cout << "ob1 == ob2\n";
    if(ob1 != ob2) cout << "error\n";
    cout << endl;

    cout << "Assign ob1[-1] the value 99\n";
    ob1[-1] = 99;
    cout << "Contents of ob1 are now:\n";
    for(i=-3; i < 3; i++)
        cout << ob1[i] << " ";
    cout << endl;

    if(ob1 == ob2) cout << "error\n";
    if(ob1 != ob2) cout << "ob1 != ob2\n";
    cout << endl;

    if(ob1 < ob2) cout << "ob1 < ob2\n";
```

```

if(ob1 <= ob2) cout << "ob1 <= ob2\n";
if(ob1 > ob2) cout << "ob1 > ob2\n";
if(ob1 >= ob2) cout << "ob1 >= ob2\n";

if(ob2 < ob1) cout << "ob2 < ob1\n";
if(ob2 <= ob1) cout << "ob2 <= ob1\n";
if(ob2 > ob1) cout << "ob2 > ob1\n";
if(ob2 >= ob1) cout << "ob2 >= ob1\n";
cout << endl;

// Compare objects of differing sizes.
if(ob3 != ob1) cout << "ob3 != ob1\n";
if(ob3 == ob1) cout << "ob3 == ob1\n";

return 0;
}

```

其输出如下所示:

```

Contents of ob1 and ob2:
-3 -2 -1 0 1 2
-3 -2 -1 0 1 2

ob1 == ob2

Assign ob1[-1] the value 99
Contents of ob1 are now:
-3 -2 99 0 1 2
ob1 != ob2

ob1 > ob2
ob1 >= ob2
ob2 < ob1
ob2 <= ob1

ob3 != ob1

```

第三个程序将类对象存储在 `RangeArray` 中。这个程序还说明了当发生不同的操作时，调用构造函数和析构函数的时刻：

```

// Store class objects in a RangeArray.
#include <iostream>
#include "ra.h"

using namespace std;

class test {
public:
    int a;

    test() { cout << "Constructing\n"; a=0; }

```

```

test(const test &o) {
    cout << "Copy Constructor\n";
    a = o.a;
}

~test() { cout << "Destructing\n"; }
};

int main()
{
    RangeArray<test> t(-3, 1, test());
    int i;

    cout << "Original contents of t:\n";
    for(i=-3; i < 2; i++) cout << t[i].a << " ";
    cout << endl;

    // Give t some new values.
    for(i=-3; i < 2; i++) t[i].a = i;

    cout << "New contents of t:\n";
    for(i=-3; i < 2; i++) cout << t[i].a << " ";
    cout << endl;

    // Copy to new container.
    RangeArray<test> t2(-7, 3, test());
    copy(t.begin(), t.end(), &t2[-2]);

    cout << "Contents of t2:\n";
    for(i=-7; i < 4; i++) cout << t2[i].a << " ";
    cout << endl;

    RangeArray<test> t3(t.begin()+1, t.end()-1);
    cout << "Contents of t3:\n";
    for(i=t3.getlowerbound(); i < t3.getupperbound(); i++)
        cout << t3[i].a << " ";
    cout << endl;

    t.clear();

    cout << "Size after clear(): " << t.size() << endl;

    // Assign container objects.
    t = t3;
    cout << "Contents of t:\n";
    for(i=t.getlowerbound(); i < t.getupperbound(); i++)
        cout << t[i].a << " ";
    cout << endl;

    return 0;
}

```


这个程序的输出如下所示:

[illegible]

```
Destructing  
Destructing  
Destructing  
Destructing  
Destructing  
Destructing
```

8.4.5 尝试完成以下任务

您可能想要完善并试验 `RangeArray` 类。例如，您可以加入反向迭代器以及 `rbegin()` 和 `rend()` 函数。在此还有一些其他的想法，您可以试着优化这个容器。如前所述，`RangeArray` 中的代码是为了操作的透明而不是速度设计的，因此可以轻易地提升速度。例如，当创建对象时，试着分配比所需多一点的内存，从而并不是所有的插入操作都强制重新分配。试着创建一个成员函数将 `RangeArray` 转换为标准的、基数为 0 的数组。试着加入采用标准数组和索引作为参数的构造函数，让这个构造函数将数组转换为 `RangeArray`，将这个索引作为 0 的位置。最后，试着使用一个 `vector` 而不是标准的数组来保存 `RangeArray` 的元素。观察它是否简化了实现(您将会得到一个惊喜)。

第 9 章 Mini C++ 解释程序

前面的章节已经揭示了 C++ 语言的丰富与强大，并描述了可以应用它的广泛任务。本章作为本书的最后一章，将从不同的观点来审视 C++：设计的艺术。C++ 的强大功能是建立在优雅的、逻辑上一致的结构基础上的。正是这种坚实的架构将 C++ 的各个特性凝聚为一个有聚合力的整体。

为了探索 C++ 优雅的设计，本章开发了一个程序来理解这门语言。在此有两种类型的程序可以做到这一点：编译器和解释程序。尽管为 C++ 创建一个编译器远远超出了本书的范围，但创建一个解释程序却是比较容易管理的任务。因此，本章为 C++ 的一个子集开发了一个解释程序。这个解释程序称为 Mini C++。由于 Mini C++ 解释 C++ 语言，因此它的实现用具体的示例说明了 C++ 语法的基本原理。

除了演示 C++ 逻辑上一致的本性之外，Mini C++ 还有另外一个优点：它给予您一个可执行的引擎，您可以按需要增强或者修改它。例如，可以扩展 Mini C++ 来解释一个更大的 C++ 子集，或者解释一种完全不同的计算机语言。甚至可以将它作为测试平台来试验您自己设计的计算机语言。它的使用是不受限制的。

9.1 解释程序和编译器

在开始之前，有必要解释一下什么是解释程序，以及它与编译器的区别。尽管编译器和解释程序都是将程序的源代码作为输入，但是它们处理源代码的方式却有很大的区别。

编译器将程序的源代码转换为一种可以执行的形式。通常，例如在 C++ 中，这种可执行的形式由 CPU 指令组成，CPU 指令可以直接由计算机执行。在其他情况下，编译器的输出是一种可移植的中间形式，这种形式可以被运行系统执行。例如，Java 会编译一种称为字节码的中间形式，字节码由 Java 虚拟机执行。

解释程序以完全不同的方式运行。它将源代码读入程序，并执行碰到的每一条语句。因此，解释程序不会将源代码转换为可执行代码。相反，解释程序直接执行这个程序。解释方式的主要缺点是降低了执行速度。以解释方式运行的程序要比编译形式慢很多。

有时术语“解释程序”会用在不同于刚才描述的情况下。例如，最初的 Java 虚拟机被称为“字节码解释程序”。这与本章所开发的解释程序不是一种类型。在此开发的解释程序是“源代码”解释程序，意味着它通过简单地阅读源代码来执行程序。

尽管编译的程序比解释的程序执行快，但是解释程序在程序设计中的应用仍然很普遍，原因如下。首先，解释程序提供了一个真正的交互式环境，其中程序可以由用户的交互来暂停或者重新启动。举例来说，这种交互式环境很适合于机器人技术。其次，由于语言解释程序的本性，它们特别适合于交互式调试。第三，解释程序对于“脚本语言”（如数据库的查询语言）很优秀。第四，它们允许在不同的平台上运行相同的程序。每个新的环境只需要实现解释程序的

运行包。

还有一个原因使得解释程序很有趣：它们易于修改、变更或者增强。这意味着如果您想要创建、试验或者控制您自己的语言，那么使用解释程序要比使用编译器容易。解释程序创建了一个了不起的语言原型环境，因为您可以改变语言，并很快看到结果。

当然，对于本章的目的而言，解释程序的最大好处是使得您能够直接看到 C++ 的内部运行。

9.2 Mini C++纵览

Mini C++ 由许多代码组成。在开始之前，大体上理解 Mini C++ 的组织方式是有好处的。Mini C++ 包含了 3 个主要的子系统：处理数学表达式的表达式解析器；实际执行程序的解释程序；还有一组库函数。

表达式解析器求解数学表达式的值，并返回结果。表达式可以包含常量，如 10 或者 88，变量、函数调用，当然还包含运算符。例如：

```
x+num/32
```

是一个表达式。

解释程序模块执行 C++ 程序。解释程序通过每次读入源代码的一个令牌来运行。当它遇到关键字时，会满足关键字的任何请求。例如，当解释程序读取 if 时，就会处理 if 指定的条件。如果这个条件为 true，则解释程序执行与 if 相关的代码块。解释程序一直运行，直到程序结束。

库子系统定义了 Mini C++ 中内建的函数。因此，它们组成了 Mini C++ 的“标准库”。在本章只包含了少数的库函数，但是只要您愿意，就可以加入任意数量的其他函数。总的来说，Mini C++ 的库函数与真正的 C++ 库函数的目的相同。

Mini C++ 将这些子系统组织到 3 个文件中，如表 9-1 所示。

表 9-1 子系统组织到文件中

子 系 统	文 件 名
解析器	parser.cpp
解释程序	minicpp.cpp
库	libcpp.cpp

Mini C++ 还使用了头文件 mcommon.h。

9.3 Mini C++说明

尽管 C++ 的关键字相对比较少，但它仍然是一门丰富而复杂的语言。描述并实现一个针对整个 C++ 的解释程序远远超出了本章的范围。取而代之的是，Mini C++ 只能理解这门语言相当有限的子集。然而，这个特定的子集包含了许多 C++ 最重要的特性。例如，Mini C++ 支持递归函数、全局和局部变量、嵌套的作用域以及大多数的控制语句。Mini C++ 支持的所有特性的列表如下所示：

- 具有局部变量的参数化函数

- 嵌套的作用域
- 递归
- if 语句
- switch 语句
- do-while、while 以及 for 循环
- break 语句
- int、char 类型的局部和全局变量
- 整型和字符型的函数参数
- 整型和字符型常量
- 字符串常量(部分实现)
- return 语句, 可以带返回值, 也可以不带返回值。
- 少量的标准库函数
- 运算符+、-、*、/、%、<、>、<=、>=、==、!=、++、--、一元- 以及一元+
- 返回整型数的函数
- “/*” 和 “//” 注释
- 使用 cin 和 cout 的控制台 I/O

虽然这个列表看起来很短, 但是实现它却需要许多代码。原因之一是当解释像 C++ 这样的语言时, 必须支付许多的“入场费用”。尽管刚才列出的特性只是 C++ 语言的一个小子集, 但它们确实使得这个解释程序能够处理这门语言的核心, 包括基础语法、控制语句、表达式以及函数调用机制。因此, Mini C++ 处理了语言的“骨干”。

毫无疑问, 您注意到了 Mini C++ 不支持类。这样做的原因只是考虑到实际情况。支持类意味着解释程序也支持用户自定义类型、对象实例化以及点(.)运算符。另外, 类要求解释程序理解 public 和 private。尽管解释程序处理这些特性本身并不困难, 然而解释程序的代码将会比本章所显示的大的多。一旦您理解了解释程序的运行方式, 您或许会发现亲自加入对类的支持很有趣。

另一些没有支持的特性包括函数以及运算符的重载、模板、命名空间、异常、预处理器、结构、联合以及位运算。同样, 解释这些特性并不困难, 但是这样做会使得代码不可预知地增长, 从而不能以书本的形式来使用。然而, 加入对这些特性的支持确实是一个有趣的项目, 您或许想亲自体验它。

Mini C++的一些限制

虽然只实现了 C++ 的一个小子集, 但是 Mini C++ 的源代码相当长。为了阻止它变得更长, 对 C++ 程序员强加了一些限制。首先, if、while、do 以及 for 的目标必须是用左括号和右括号包围起来的代码块。不能使用一条单个的语句。例如, Mini C++ 将不会正确地解释如下的代码:

```
for(a=0; a < 10; a++)
    for(b=0; b < 10; b++)
        for(c=0; c < 10; c++)
            cout << "hi";

if(...)
    if(...) x = 10;
```

取而代之的是，您必须以这种方式来编写代码：

```
for(a=0; a < 10; a++) {
    for(b=0; b < 10; b++) {
        for(c=0; c < 10; c++) {
            cout << "hi";
        }
    }
}

if(...) {
    if(...) {
        x = 10;;
    }
}
```

这些限制使得解释程序能够比较容易地发现构成这些程序控制语句目标的代码结尾。然而，由于这些程序控制语句的对象经常是代码块，因此这些限制并不是很苛刻。(如果您愿意的话，只需要一点点功夫，就可以去除这些限制)。

另一个限制是不支持原型。所有的函数都被假定返回一个整型值(也允许返回字符型，但是会转化为整型)，并且不会执行参数类型的检查。另外，由于不支持函数重载，因此函数必须有且仅有一个版本。

除了不支持 default 语句外，switch 语句完全实现。这个限制是为了降低解释 switch 语句所需代码的大小和复杂程度。(switch 语句是解释起来很复杂的语句之一)。对 default 的支持需要您自己来尝试。

最后，所有的函数前面都必须使用 int 或者 char 类型说明符。因此，Mini C++解释程序不支持 void 返回类型。从而，下面的声明是有效的：

```
int f()
{
    // ...
}
```

但是这个声明是无效的：

```
void f()
{
    // ...
}
```

9.4 非正式的 C++理论

为了充分理解 Mini C++的操作，有必要理解 C++语言的构造方式。如果您已经看到过 C++语言的正式说明(如 ANSI/ISO C++标准中的内容)，就知道这些说明非常长，并且充满了相当晦涩的语句。不要着急——您不需要正式地学习 C++语言的说明来理解 Mini C++。Mini C++解释的那部分 C++很容易理解。也就是说，您对 C++语言的定义方式只需要有基本的了解。对于我们的目的而言，下面非正式的讨论就足够了。然而要记住，这些讨论有意简化了一些概念。

C++程序由一个或者多个函数的集合以及全局变量(如果有的话)组成。函数由函数名、参数列表以及函数体组成,函数体是一个代码块。代码块以“{”开始,随后是一条或者多条语句,并且以“}”结束。代码块(也称为复合语句)创建了一个作用域。通常,语句或者是表达式、嵌套的代码块,或者以关键字(如 if、for 或者 int)开始。嵌套的代码块创建了嵌套的作用域。

提示:

尽管前面对语句(关键字、代码块以及表达式)的分类对于理解 Mini C++已经足够,但是 C++的 ANSI/ISO 标准使用了更细粒度的定义。它将语句定义为可标记的(labeled)、表达式、复合、选择、迭代、跳转、声明以及 try 块。

C++程序通过调用 main()开始运行。当在 main()中遇到最后的“}”或者 return 时,程序结束——假定在其他地方没有调用 exit()或者 abort()。程序中的其他函数都必须直接或者间接地被 main()函数调用。因此,当 Mini C++运行一个程序时,只是简单地从 main()函数开始并且在 main()函数结束时停止。

9.4.1 C++表达式

C++的基础是表达式,因为在程序中的许多操作都是通过表达式发生的。为了理解原因,回忆一下,C++有三类主要的语句:关键字、语句块以及表达式。这意味着按照定义,任何不是基于关键字的语句或者没有定义语句块的语句都是表达式语句。因此,下面的语句都是表达式:

```
count = 100; // line 1
sample = i / 22 * (c-10); // line 2
num = abs(count) * 2; // line 3
strcpy(str1, str2); // line 4
```

让我们仔细观察每一条表达式语句。在 C++中,等号是一个赋值运算符。这很重要。C++没有像某些语言那样,把赋值当作单独的语句类型。相反,等号是一种赋值运算符,赋值操作产生的值等于右边表达式产生的值。因此在 C++中,一条赋值语句实际上是一个赋值表达式。由于它是表达式,因此具有值。这就是下面的表达式合法的原因:

```
a = b = c = 100;
if( (a=4+5) == 0 ) ...;
```

这些表达式可以运行的原因是赋值是一个可以产生值的操作。

第 2 行显示了一个更加复杂的赋值表达式。在此情况下,计算右边表达式的值,并赋给 sample。

在第 3 行,调用了 abs()来返回参数的绝对值。因此,表达式中使用函数会导致调用这个函数。

正如第 4 行所示的那样,函数调用本身就是表达式。在此情况下,调用了 strcpy()来将一个字符串的内容复制到另一个字符串。虽然 strcpy()的返回值被忽略,但是 strcpy()的调用仍然构成一个表达式。

9.4.2 定义表达式

为了对表达式求值,Mini C++使用了一个表达式解析器。在理解表达式解析器的操作之前,

需要了解表达式的构造方法。在几乎所有的计算机语言中，都是用一组产生式规则(production rule)来描述表达式的，产生式规则定义了操作数与运算符的交互。定义了 Mini C++表达式的产生式规则如图 9-1 所示。

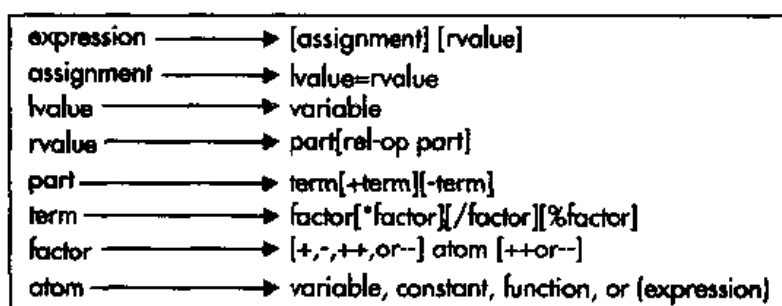


图 9-1 Mini C++表达式的产生式规则

在此，rel-op 指任何的 C++关系运算符。术语 lvalue 和 rvalue 分别指赋值语句左边和右边的对象。箭头指“产生”。有一点需要知道，运算符的优先级内建于产生式规则中。优先级越高，运算符在列表中的位置就越靠下。正如产生式规则所显示的那样，Mini C++支持如下的运算符：

+ - * / % ++ --
 < <= > >= == != ()

为了观察这些规则的运行方式，对如下的 C++表达式求值：

count=10-5*3;

首先，应用规则 1，将这个表达式分解为下面 3 个部分。见图 9-2。



图 9-2 表达式的 3 个部分

由于在分解后表达式的“rvalue”部分没有关系运算符，因此应用了术语产生式规则。参见图 9-3。

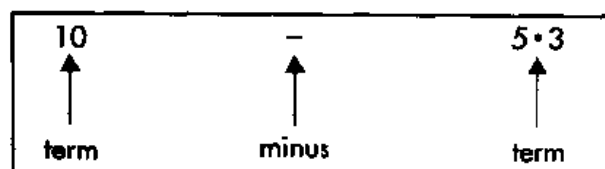


图 9-3 产生式规则

当然，第 2 项由因子 3 和 5 组成。这两个因子是常量，代表了产生式规则的最底层。随后，再次回到这个规则来计算表达式的值。首先，5*3 等于 15。然后，用 10 减这个值，得到-5。最后，这个值赋给 count，这也是整个表达式的值。Mini C++表达式解析器以相同的方式求解这个表达式。它简单地实现了产生式规则。

由于表达式解析器对 Mini C++非常重要，因此下面我们来详细介绍表达式解析器。

9.5 表达式解析器

读取并分析表达式的代码段称为表达式解析器。毫无疑问，表达式解析器是 Mini C++ 需要的最重要的子系统。由于 C++ 定义的表达式比许多其他语言广泛的多，因此表达式解析器执行了解组成 C++ 程序的大量代码。

可以通过多种不同的方法来为 C++ 设计一个表达式解析器。许多商业编译器使用了表格驱动的解析器，这个解析器通常由解析器生成器程序创建。尽管表格驱动的解析器通常比其他方法要快，但是它们很难手工创建。Mini C++ 使用了递归下降解析器(recursive-descent parser)，这个解析器在逻辑上实现了刚才讨论的产生式规则。

递归下降解析器本质上是相互递归函数的集合，此集合处理表达式。如果这个解析器在编译器中使用，那么会生成对应于源代码的正确的对象代码。然而在解释程序中，解析器求解给定表达式的值。在这个部分，开发了 Mini C++ 解析器。

提示：

作者编写范围广泛的表达式解析器已经很多年了。当涉及到 C++ 时，对这个问题的深入观察请参考作者的 C++ 书籍：*The Complete Reference*, McGraw-Hill/Osborne。

9.5.1 解析器代码

Mini C++ 递归下降解析器的整个代码如下所示。这些代码保存在文件 `parser.cpp` 中。后面部分将解释这个解析器的操作。

```
// Recursive descent parser for integer expressions.
//
#include <iostream>
#include <cstring>
#include <cstdlib>
#include <cctype>
#include "mccommon.h"

using namespace std;

// Keyword lookup table.
// Keywords must be entered lowercase.
struct commands {
    char command[20];
    token_ireps tok;
} com_table[] = {
    "if", IF,
    "else", ELSE,
    "for", FOR,
    "do", DO,
    "while", WHILE,
    "char", CHAR,
    "int", INT,
    "return", RETURN,
    "switch", SWITCH,
```

```

    "break", BREAK,
    "case", CASE,
    "cout", COUT,
    "cin", CIN,
    "", END // mark end of table
};

// This structure links a library function name
// with a pointer to that function.
struct intern_func_type {
    char *f_name; // function name
    int (*p)(); // pointer to the function
} intern_func[] = {
    "getchar", call_getchar,
    "putchar", call_putchar,
    "abs", call_abs,
    "rand", call_rand,
    "", 0 // null terminate the list
};

// Entry point into parser.
void eval_exp(int &value)
{
    get_token();

    if(!*token) {
        throw InterpExc(NO_EXP);
    }

    if(*token == ';') {
        value = 0; // empty expression
        return;
    }
    eval_exp0(value);

    putback(); // return last token read to input stream
}

// Process an assignment expression.
void eval_exp0(int &value)
{
    // temp holds name of var receiving the assignment.
    char temp[MAX_ID_LEN+1];

    tok_types temp_tok;

    if(token_type == IDENTIFIER) {
        if(is_var(token)) { // if a var, see if assignment
            strcpy(temp, token);
            temp_tok = token_type;
            get_token();
            if(*token == '=') { // is an assignment

```

```

    get_token();
    eval_exp0(value); // get value to assign
    assign_var(temp, value); // assign the value
    return;
}
else { // not an assignment
    putback(); // restore original token
    strcpy(token, temp);
    token_type = temp_tok;
}
}
}
eval_exp1(value);
}

// Process relational operators.
void eval_exp1(int &value)
{
    int partial_value;
    char op;
    char relops[] = {
        LT, LE, GT, GE, EQ, NE, 0
    };

    eval_exp2(value);
    op = *token;
    if(strchr(relops, op)) {
        get_token();
        eval_exp2(partial_value);

        switch(op) { // perform the relational operation
            case LT:
                value = value < partial_value;
                break;
            case LE:
                value = value <= partial_value;
                break;
            case GT:
                value = value > partial_value;
                break;
            case GE:
                value = value >= partial_value;
                break;
            case EQ:
                value = value == partial_value;
                break;
            case NE:
                value = value != partial_value;
                break;
        }
    }
}
}

```

```

// Add or subtract two terms.
void eval_exp2(int &value)
{
    char op;
    int partial_value;
    char okops[] = {
        '(', INC, DEC, '-', '+', 0
    };

    eval_exp3(value);

    while((op = *token) == '+' || op == '-') {
        get_token();

        if(token_type == DELIMITER &&
            !strchr(okops, *token))
            throw InterpExc(SYNTAX);
        eval_exp3(partial_value);

        switch(op) { // add or subtract
            case '-':
                value = value - partial_value;
                break;
            case '+':
                value = value + partial_value;
                break;
        }
    }
}

```

```

// Multiply or divide two factors.
void eval_exp3(int &value)
{
    char op;
    int partial_value, t;
    char okops[] = {
        '(', INC, DEC, '-', '+', 0
    };

    eval_exp4(value);

    while((op = *token) == '*' || op == '/'
        || op == '%') {
        get_token();

        if(token_type == DELIMITER &&
            !strchr(okops, *token))
            throw InterpExc(SYNTAX);

        eval_exp4(partial_value);
    }
}

```

```

switch(op) { // mul, div, or modulus
    case '*':
        value = value * partial_value;
        break;
    case '/':
        if(partial_value == 0)
            throw InterpExc(DIV_BY_ZERO);
        value = (value) / partial_value;
        break;
    case '%':
        t = (value) / partial_value;
        value = value - (t * partial_value);
        break;
}
}
}

// Is a unary +, -, ++, or --.
void eval_exp4(int &value)
{
    char op;
    char temp;

    op = '\0';
    if(*token == '+' || *token == '-' ||
        *token == INC || *token == DEC)
    {
        temp = *token;
        op = *token;
        get_token();
        if(temp == INC)
            assign_var(token, find_var(token)+1);
        if(temp == DEC)
            assign_var(token, find_var(token)-1);
    }

    eval_exp5(value);
    if(op == '-') value = -(value);
}

// Process parenthesized expression.
void eval_exp5(int &value)
{
    if((*token == '(')) {
        get_token();

        eval_exp0(value); // get subexpression

        if(*token != ')')
            throw InterpExc(PAREN_EXPECTED);
        get_token();
    }
}

```

```

    }
    else
        atom(value);
}

// Find value of number, variable, or function.
void atom(int &value)
{
    int i;
    char temp[MAX_ID_LEN+1];

    switch(token_type) {
        case IDENTIFIER:
            i = internal_func(token);
            if(i != -1) {
                // Call "standard library" function.
                value = (*intern_func[i].p)();
            }
            else if(find_func(token)) {
                // Call programmer-created function.
                call();
                value = ret_value;
            }
            else {
                value = find_var(token); // get var's value
                strcpy(temp, token); // save variable name

                // Check for ++ or --.
                get_token();
                if(*token == INC || *token == DEC) {
                    if(*token == INC)
                        assign_var(temp, find_var(temp)+1);
                    else
                        assign_var(temp, find_var(temp)-1);
                } else putback();
            }

            get_token();
            return;
        case NUMBER: // is numeric constant
            value = atoi(token);
            get_token();

            return;
        case DELIMITER: // see if character constant
            if(*token == '\\') {
                value = *prog;
                prog++;
                if(*prog != '\\')
                    throw InterpExc(QUOTE_EXPECTED);
                prog++;
                get_token();
            }
    }
}

```

```

        return ;
    }
    if(*token=='') return; // process empty expression
    else throw InterpExc(SYNTAX); // otherwise, syntax error
default:
    throw InterpExc(SYNTAX); // syntax error
}
}

// Display an error message.
void sntx_err(error_msg error)
{
    char *p, *temp;
    int linecount = 0;

    static char *e[] = {
        "Syntax error",
        "No expression present",
        "Not a variable",
        "Duplicate variable name",
        "Duplicate function name",
        "Semicolon expected",
        "Unbalanced braces",
        "Function undefined",
        "Type specifier expected",
        "Return without call",
        "Parentheses expected",
        "While expected",
        "Closing quote expected",
        "Division by zero",
        "{ expected (control statements must use blocks)",
        "Colon expected"
    };

    // Display error and line number.
    cout << "\n" << e[error];
    p = p_buf;
    while(p != prog) { // find line number of error
        p++;
        if(*p == '\r') {
            linecount++;
        }
    }
    cout << " in line " << linecount << endl;

    temp = p;
    while(p > p_buf && *p != '\n') p--;

    // Display offending line.
    while(p <= temp)
        cout << *p++;

```

```

    cout << endl;
}

// Get a token.
tok_types get_token()
{
    char *temp;

    token_type = UNDEFTT; tok = UNDEFTOK;

    temp = token;
    *temp = '\0';

    // Skip over white space.
    while(isspace(*prog) && *prog) ++prog;

    // Skip over newline.
    while(*prog == '\r') {
        ++prog;
        ++prog;
        // Again, skip over white space.
        while(isspace(*prog) && *prog) ++prog;
    }

    // Check for end of program.
    if(*prog == '\0') {
        *token = '\0';
        tok = END;
        return (token_type = DELIMITER);
    }

    // Check for block delimiters.
    if(strchr("{} ", *prog)) {
        *temp = *prog;
        temp++;
        *temp = '\0';
        prog++;
        return (token_type = BLOCK);
    }

    // Look for comments.
    if(*prog == '/')
        if(*(prog+1) == '*') { // is a /* comment
            prog += 2;
            do { // find end of comment
                while(*prog != '*') prog++;
                prog++;
            } while (*prog != '/');
            prog++;
            return (token_type = DELIMITER);
        }
    }

```



```

    } else if(*(prog+1) == '/') { // is a // comment
        prog += 2;
        // Find end of comment.
        while(*prog != '\r' && *prog != '\0') prog++;
        if(*prog == '\r') prog += 2;
        return (token_type = DELIMITER);
    }

// Check for double-ops.
if(strchr("!<>=+-", *prog)) {
    switch(*prog) {
        case '=':
            if(*(prog+1) == '=') {
                prog++; prog++;
                *temp = EQ;
                temp++; *temp = EQ; temp++;
                *temp = '\0';
            }
            break;
        case '!':
            if(*(prog+1) == '=') {
                prog++; prog++;
                *temp = NE;
                temp++; *temp = NE; temp++;
                *temp = '\0';
            }
            break;
        case '<':
            if(*(prog+1) == '=') {
                prog++; prog++;
                *temp = LE; temp++; *temp = LE;
            }
            else if(*(prog+1) == '<') {
                prog++; prog++;
                *temp = LS; temp++; *temp = LS;
            }
            else {
                prog++;
                *temp = LT;
            }
            temp++;
            *temp = '\0';
            break;
        case '>':
            if(*(prog+1) == '=') {
                prog++; prog++;
                *temp = GE; temp++; *temp = GE;
            } else if(*(prog+1) == '>') {
                prog++; prog++;
                *temp = RS; temp++; *temp = RS;
            }
            else {

```

```

        prog++;
        *temp = GT;
    }
    temp++;
    *temp = '\0';
    break;
case '+':
    if(*(prog+1) == '+') {
        prog++; prog++;
        *temp = INC; temp++; *temp = INC;
        temp++;
        *temp = '\0';
    }
    break;
case '-':
    if(*(prog+1) == '-') {
        prog++; prog++;
        *temp = DEC; temp++; *temp = DEC;
        temp++;
        *temp = '\0';
    }
    break;
}

if(*token) return(token_type = DELIMITER);
}

// Check for other delimiters.
if(strchr("+-*/%=:(),'", *prog)) {
    *temp = *prog;
    prog++;
    temp++;
    *temp = '\0';
    return (token_type = DELIMITER);
}

// Read a quoted string.
if(*prog == '"') {
    prog++;
    while(*prog != '"' && *prog != '\r' && *prog) {
        // Check for \n escape sequence.
        if(*prog == '\\') {
            if(*(prog+1) == 'n') {
                prog++;
                *temp++ = '\n';
            }
        }
    }
    else if((temp - token) < MAX_T_LEN)
        *temp++ = *prog;

    prog++;
}

```

```

    if(*prog == '\r' || *prog == 0)
        throw InterpExc(SYNTAX);
    prog++; *temp = '\0';
    return (token_type = STRING);
}

// Read an integer number.
if(isdigit(*prog)) {
    while(!isdelim(*prog)) {
        if((temp - token) < MAX_ID_LEN)
            *temp++ = *prog;
        prog++;
    }
    *temp = '\0';
    return (token_type = NUMBER);
}

// Read identifier or keyword.
if(isalpha(*prog)) {
    while(!isdelim(*prog)) {
        if((temp - token) < MAX_ID_LEN)
            *temp++ = *prog;
        prog++;
    }
    token_type = TEMP;
}

*temp = '\0';

// Determine if token is a keyword or identifier.
if(token_type == TEMP) {
    tok = look_up(token); // convert to internal form
    if(tok) token_type = KEYWORD; // is a keyword
    else token_type = IDENTIFIER;
}

// Check for unidentified character in file.
if(token_type == UNDEFTT)
    throw InterpExc(SYNTAX);

return token_type;
}

// Return a token to input stream.
void putback()
{
    char *t;

    t = token;
    for(; *t; t++) prog--;
}

```

```

// Look up a token's internal representation in the
// token table.
token_ireps look_up(char *s)
{
    int i;
    char *p;

    // Convert to lowercase.
    p = s;
    while(*p) { *p = tolower(*p); p++; }

    // See if token is in table.
    for(i=0; *com_table[i].command; i++) {
        if(!strcmp(com_table[i].command, s))
            return com_table[i].tok;
    }

    return UNDEFTOK; // unknown command
}

// Return index of internal library function or -1 if
// not found.
int internal_func(char *s)
{
    int i;

    for(i=0; intern_func[i].f_name[0]; i++) {
        if(!strcmp(intern_func[i].f_name, s)) return i;
    }
    return -1;
}

// Return true if c is a delimiter.
bool isdelim(char c)
{
    if(strchr(" !,;+-<>'/*%^=()", c) || c == 9 ||
       c == '\r' || c == 0) return true;
    return false;
}

```

这个解析器使用了下面的全局变量和枚举类型(包含在本章后面显示的头文件 `mccommon.h` 中):

```

const int MAX_T_LEN = 128; // max token length
const int MAX_ID_LEN = 31; // max identifier length

// Enumeration of token types.
enum tok_types { UNDEFTT, DELIMITER, IDENTIFIER,
                 NUMBER, KEYWORD, TEMP, STRING, BLOCK };

// Enumeration of internal representation of tokens.
enum token_ireps { UNDEFTOK, ARG, CHAR, INT, SWITCH,

```

```

CASE, IF, ELSE, FOR, DO, WHILE, BREAK,
RETURN, COUT, CIN, END };

// Enumeration of two-character operators, such as <=.
enum double_ops { LT=1, LE, GT, GE, EQ, NE, LS, RS, INC, DEC };

// These are the constants used when throwing a
// syntax error exception.
//
// NOTE: SYNTAX is a generic error message used when
// nothing else seems appropriate.
enum error_msg
{ SYNTAX, NO_EXP, NOT_VAR, DUP_VAR, DUP_FUNC,
  SEMI_EXPECTED, UNBAL_BRACES, FUNC_UNDEF,
  TYPE_EXPECTED, RET_NOCALL, PAREN_EXPECTED,
  WHILE_EXPECTED, QUOTE_EXPECTED, DIV_BY_ZERO,
  BRACE_EXPECTED, COLON_EXPECTED };

extern char *prog; // current location in source code
extern char *p_buf; // points to start of program buffer

extern char token[MAX_T_LEN+1]; // string version of token
extern tok_types token_type; // contains type of token
extern token_ireps tok; // internal representation of token

extern int ret_value; // function return value

// Exception class for Mini C++.
class InterpExc {
    error_msg err;
public:
    InterpExc(error_msg e) { err = e; }
    error_msg get_err() { return err; }
};

```

`prog` 指向源代码中当前正在执行的位置。因此，`prog` 保存了一个地址，解释程序将在这个位置读取下一块程序。解释程序不会改变 `p_buf` 指针，这个指针总是指向被解释程序的开始。当前的令牌存储在 `token` 中。(令牌是程序代码中不可分割的部分。下面部分将对其进行更精确的定义)。令牌的类型存储在 `token_type` 中。如果这个令牌是关键字，那么 `tok` 变量保存这个令牌的内部形式。

枚举类型 `tok_types` 声明了 Mini C++ 能够识别的令牌的类型。`token_ireps` 枚举类型指定了代表关键字的令牌的内部格式。代表由两个字符组成的运算符(例如 `<=`)的值由 `double_ops` 枚举类型指定。`error_msg` 枚举了各种错误代码。最后，`InterpExc` 是用来报告语法错误的异常类。

9.5.2 分解源代码

将源代码分解为组成部件的机制是所有解释程序(以及编译器，在此情况下)的基础，这些组成部件称为令牌。令牌是程序不可分割的部分，代表一个逻辑单元。例如，`{`、`+`、`=`、以及 `if` 都是令牌。虽然等号运算符 `"=="` 由两个字符组成，但是它们不能被分割，否则就会改变符号

含义。因此，`==`是一个独立的逻辑单元，从而是一个令牌。同样，`if`也是一个令牌，但是`i`或者`f`对于C++都没有特殊的含义。

C++的ANSI/ISO标准定义了下面类型的令牌：

关键字	标识符	字面值
运算符	标点	

关键字是组成C++语言的那些令牌，例如`while`。标识符是变量、函数等的名称。字面值包含字符串和常量值，如数字`10`。运算符含义自明。标点包括分号、逗号、花括号和圆括号。（某些标点符号也是运算符，取决于它们的使用）。

尽管Mini C++可以识别的令牌与ANSI/ISO C++标准指定的令牌相同，但是识别的方式却有很大的区别，目的是使得解释容易一点。Mini C++使用的令牌分类如表9-2所示。

表 9-2 Mini C++使用的令牌分类

令 牌 类 型	包 含 内 容
分隔符	标点和运算符
关键字	关键字
字符串	引用字符串
标识符	变量和函数名称
数值	数值常量
代码块	{or}

让我们完成一个示例，来说明这些令牌类型的应用方式。给定下面的语句：

```
for(x=0; x<10; x++) {
    num = num + x;
}
```

生成了如表9-3所示的令牌。

表 9-3 生成的令牌

令 牌	类 别
<code>for</code>	关键字
<code>(</code>	分隔符
<code>x</code>	标识符
<code>=</code>	分隔符
<code>0</code>	数值
<code>;</code>	分隔符
<code>x</code>	标识符
<code><</code>	分隔符
<code>10</code>	数值
<code>;</code>	分隔符
<code>x</code>	标识符

(续表)

令 牌	类 别
++	分隔符
)	分隔符
{	代码块
num	标识符
=	分隔符
num	标识符
+	分隔符
x	标识符
;	分隔符
}	代码块

get_token()函数为 Mini C++解释程序从源代码中返回令牌代码，如下所示：

```
// Get a token.
tok_types get_token()
{
    char *temp;

    token_type = UNDEFTT; tok = UNDEFTOK;

    temp = token;
    *temp = '\0';

    // Skip over white space.
    while(isspace(*prog) && *prog) ++prog;

    // Skip over newline.
    while(*prog == '\r') {
        ++prog;
        ++prog;
        // Again, skip over white space.
        while(isspace(*prog) && *prog) ++prog;
    }

    // Check for end of program.
    if(*prog == '\0') {
        *token = '\0';
        tok = END;
        return (token_type = DELIMITER);
    }

    // Check for block delimiters.
    if(strchr("{} ", *prog)) {
```

```

    *temp = *prog;
    temp++;
    *temp = '\0';
    prog++;
    return (token_type = BLOCK);
}

// Look for comments.
if(*prog == '/')
    if(*(prog+1) == '*') { // is a /* comment
        prog += 2;
        do { // find end of comment
            while(*prog != '*') prog++;
            prog++;
        } while (*prog != '/');
        prog++;
        return (token_type = DELIMITER);
    } else if(*(prog+1) == '/') { // is a // comment
        prog += 2;
        // Find end of comment.
        while(*prog != '\r' && *prog != '\0') prog++;
        if(*prog == '\r') prog += 2;
        return (token_type = DELIMITER);
    }

// Check for double-ops.
if(strchr("<=>+-", *prog)) {
    switch(*prog) {
        case '=':
            if(*(prog+1) == '=') {
                prog++; prog++;
                *temp = EQ;
                temp++; *temp = EQ; temp++;
                *temp = '\0';
            }
            break;
        case '!':
            if(*(prog+1) == '=') {
                prog++; prog++;
                *temp = NE;
                temp++; *temp = NE; temp++;
                *temp = '\0';
            }
            break;
        case '<':
            if(*(prog+1) == '=') {
                prog++; prog++;
                *temp = LE; temp++; *temp = LE;
            }
            else if(*(prog+1) == '<') {
                prog++; prog++;
                *temp = LS; temp++; *temp = LS;
            }
    }
}

```



```

    }
    else {
        prog++;
        *temp = LT;
    }
    temp++;
    *temp = '\0';
    break;
case '>':
    if(*(prog+1) == '=') {
        prog++; prog++;
        *temp = GE; temp++; *temp = GE;
    } else if(*(prog+1) == '>') {
        prog++; prog++;
        *temp = RS; temp++; *temp = RS;
    }
    else {
        prog++;
        *temp = GT;
    }
    temp++;
    *temp = '\0';
    break;
case '+':
    if(*(prog+1) == '+') {
        prog++; prog++;
        *temp = INC; temp++; *temp = INC;
        temp++;
        *temp = '\0';
    }
    break;
case '-':
    if(*(prog+1) == '-') {
        prog++; prog++;
        *temp = DEC; temp++; *temp = DEC;
        temp++;
        *temp = '\0';
    }
    break;
}

if(*token) return(token_type = DELIMITER);
}

// Check for other delimiters.
if(strchr("+-*^/%=;:(),'", *prog)) {
    *temp = *prog;
    prog++;
    temp++;
    *temp = '\0';
    return (token_type = DELIMITER);
}

```

```

// Read a quoted string.
if(*prog == '"') {
    prog++;
    while(*prog != '"' && *prog != '\r' && *prog) {
        // Check for \n escape sequence.
        if(*prog == '\\') {
            if(*(prog+1) == 'n') {
                prog++;
                *temp++ = '\n';
            }
        }
        else if((temp - token) < MAX_T_LEN)
            *temp++ = *prog;
        prog++;
    }
    if(*prog == '\r' || *prog == 0)
        throw InterpExc(SYNTAX);
    prog++; *temp = '\0';
    return (token_type = STRING);
}

// Read an integer number.
if(isdigit(*prog)) {
    while(!isdelim(*prog)) {
        if((temp - token) < MAX_ID_LEN)
            *temp++ = *prog;
        prog++;
    }
    *temp = '\0';
    return (token_type = NUMBER);
}

// Read identifier or keyword.
if(isalpha(*prog)) {
    while(!isdelim(*prog)) {
        if((temp - token) < MAX_ID_LEN)
            *temp++ = *prog;
        prog++;
    }
    token_type = TEMP;
}

*temp = '\0';

// Determine if token is a keyword or identifier.
if(token_type == TEMP) {
    tok = look_up(token); // convert to internal form
    if(tok) token_type = KEYWORD; // is a keyword
    else token_type = IDENTIFIER;
}

```

```

    // Check for unidentified character in file.
    if(token_type == UNDEFTT)
        throw InterpExc(SYNTAX);

    return token_type;
}

```

`get_token()`函数开始忽略所有的空白，包括回车和换行。由于令牌不会包含空格(除非是被引用的字符串或者字符常量)，因此必须忽略空格。`get_token()`函数还忽略了所有的注释。然后读取程序中的下一个令牌。注意各种令牌的处理方式。例如，如果程序中的下一个字符是数字，就会读取一个数字。如果下一个字符是字母，就会获取一个标识符或者关键字等。

令牌的字符串表示被放入 `token` 中。在读取时，令牌的类型(`tok_types` 枚举所枚举的那些)被放入 `token_type`，如果令牌是一个关键字，则它的内部表示(例如 `token_ireps` 枚举的)通过 `look_up()`函数赋给 `tok`。(后面将解释关键字内部表示的原因)。正如您看到的那样，`get_token()`将两个字符的 C++运算符，如 `<=`或者 `++`，转换为对应的枚举值。尽管技术上并没有这样要求，但是这样做会使得解析器易于实现。

9.5.3 显示语法错误

如果解析器碰到语法错误，就会抛出 `InterpExc` 异常，并指定与所发现的错误类型对应的枚举值。(Mini C++的其他部分还用 `InterpExc` 来报告错误)。`InterpExc` 异常的处理程序在 `main()` 函数中，这是解释程序的主要文件 `minicpp.cpp` 的一部分，将在后面描述。这个处理程序调用了如下所示的 `sntx_err()`函数来报告这个错误：

```

// Display an error message.
void sntx_err(error_msg error)
{
    char *p, *temp;
    int linecount = 0;

    static char *e[] = {
        "Syntax error",
        "No expression present",
        "Not a variable",
        "Duplicate variable name",
        "Duplicate function name",
        "Semicolon expected",
        "Unbalanced braces",
        "Function undefined",
        "Type specifier expected",
        "Return without call",
        "Parentheses expected",
        "While expected",
        "Closing quote expected",
        "Division by zero",
        "{ expected (control statements must use blocks)",
        "Colon expected"
    };
};

```

```

// Display error and line number.
cout << "\n" << e[error];
p = p_buf;
while(p != prog) { // find line number of error
    p++;
    if(*p == '\r') {
        linecount++;
    }
}
cout << " in line " << linecount << endl;

temp = p;
while(p > p_buf && *p != '\n') p--;

// Display offending line.
while(p <= temp)
    cout << *p++;

cout << endl;
}

```

注意, `sntx_err()`显示了一个字符串来描述错误以及发现错误的那一行的编号(可能是发生错误那行的下一行)。这个函数还显示了发现错误的那一行。

9.5.4 表达式求值

以 `eval_exp` 开头的函数和 `atom()`函数实现了 Mini C++表达式的产生式规则。为了理解解析器对表达式求值的方式, 对下面的表达式进行操作。(假定 `prog` 指向表达式的开始)。

10-3*2

当调用解析器的入口函数 `eval_exp()`时, 首先获取第一个令牌。如果这个令牌为空, 就会抛出一个异常, 指示没有出现表达式。然而, 在此情况下, 令牌包含了数字 10。由于令牌不为空, 因此调用 `exval_exp0()`, 这个函数检查赋值运算符。由于没有出现赋值, 因此调用 `eval_exp1()`, 这个函数依次调用了 `eval_exp2()`。接着, `eval_exp2()`调用 `eval_exp3()`, `eval_exp3()`调用 `eval_exp4()`。`eval_exp4()`处理一元+和-, 以及前缀++和--。这个函数随后调用 `eval_exp5()`。在 `eval_exp5()`函数中, 或者递归调用 `eval_exp0()`(在使用了括号的表达式中), 或者递归调用 `atom()`找到一个值。由于这个令牌不是左括号, 因此执行 `atom()`, 给 `value` 赋值 10。随后检索另一个令牌, 函数沿着链条向上返回。现在这个令牌是运算符 -, 函数一直返回到 `eval_exp2()`。

接下来发生的事情很重要。由于这个令牌是 -, 因此将其保存在 `op` 中。然后解析器获取下一个令牌, 这个令牌是 3, 这个链条重新向下开始。与前面一样, 进入 `atom()`。3 返回到 `value` 中, 读取运算符*。从而导致链条返回到 `eval_exp3()`, 在此处读取最后的令牌 2。在这里进行了第一个算术操作——2 乘以 3。这个结果返回到 `eval_exp2()`, 并执行减运算。减运算产生了答案 4。尽管这个过程看起来很复杂, 但您亲自来完成其他的示例就会清楚解析器的操作。

现在, 让我们更加仔细地观察 `atom()`函数。这个函数寻找一个整型常量或者变量、函数或者字符常量的值。(这个函数还处理后缀++以及--)。在源代码中, 可能出现两种类型的函数: 用户自定义函数或者库函数。如果遇到了用户自定义函数, 解释程序执行它的代码来判断返回

值。(在后面的部分将讨论 Mini C++ 实际完成函数调用的方式)。如果这个函数是库函数, 首先使用函数 `internal_func()` 来查找其地址, 然后通过接口函数来访问它。库函数和接口函数的地址保存在 `intern_func` 数组中, 如下所示:

```
// This structure links a library function name
// with a pointer to that function.
struct intern_func_type {
    char *f_name; // function name
    int (*p)(); // pointer to the function
} intern_func[] = {
    "getchar", call_getchar,
    "putchar", call_putchar,
    "abs", call_abs,
    "rand", call_rand,
    "", 0 // null terminate the list
};
```

正如您所看到的那样, Mini C++ 只解释少数的库函数, 但是如果需要的话, 很容易加入其他的函数。(实际的接口函数保存在一个单独的文件中, 将在“Mini C++ 库函数”那一部分讨论)。

在表达式解析器文件中, 关于这个例程的最后一点是: 为了正确地分析 C++ 语法, 有时需要“提前查找一个令牌”。例如, 考虑下面的语句

```
alpha=count();
```

为了让 Mini C++ 知道 `count` 是一个函数而不是变量, 必须读取 `count` 和下一个令牌。如果下一个令牌是括号, 就像这里的情况, 那么解析器知道 `count` 是一个函数。然而, 如果这条语句是:

```
alpha = count*10;
```

那么 `count` 之后的下一个令牌是 `*`。由于它不是圆括号, 则意味着 `count` 一定是一个变量。如果没有提前读取下一个令牌, 则解析器没有办法作出判断。在两种情况下, 都需要把提前读取的令牌返回到输入流, 以进行后面的处理。为此, 表达式解析器文件包含了 `putback()` 函数, 这个函数返回从输入流读取的最后一个令牌。

9.6 Mini C++ 解释程序

实际执行 C++ 程序的引擎是解释程序。在进入代码之前, 理解解释程序一般的操作方式是有好处的。在许多方面, 解释程序的代码比表达式解析器的代码容易理解, 因为在概念上, 解释 C++ 程序的动作可以总结为以下的算法:

```
当(令牌存在){
    获取下一个令牌;
    采取正确的动作;
}
```

与表达式解析器相比, 这个算法不可思议地简单, 但是所有的解释程序确实都是这么做的。

需要记住的一点是，“采取正确的动作”步骤可能涉及到从输入流而来的其他令牌，并且需要进一步的正确动作。因此，“采取正确的动作”步骤可以是递归的。

为了理解这个算法确切的运行方式，让我们手工解释下面的代码段：

```
int a;  
  
a = 10 * num;  
  
if(a < 100) {  
    cout << a;  
}
```

按照这个算法，读取第一个令牌 `int`。对这个令牌执行的恰当动作是读取下一个令牌，以找出被声明变量的名称(在此情况下是 `a`)，然后存储这个名称。下一个令牌是结束这一行的分号。在此采取的正确动作是忽略它。现在，获取下一个令牌 `a`。由于这条语句不是以关键字开始，那么它一定是一个表达式。在此，正确的动作是使用解析器求这个表达式的值。这个过程耗尽了这条语句所有的令牌。随后读取了 `if` 令牌。这表示 `if` 语句的开始，因此正确的动作是处理 `if`，这意味着求条件表达式的值。如果表达式的结果为 `true`，就会解释与 `if` 语句相关的代码块，否则就会将其忽略。

任何 C++ 程序都会发生刚才描述的过程。直到读取了最后一个令牌或者遇到了语法错误，解释过程才会停止。记住基本的算法，让我们检查这个解释程序。整个的解释程序文件名为 `minicpp.cpp`，如下所示。随后的部分将详细分析这些代码。

```
// A Mini C++ interpreter.  
  
#include <iostream>  
#include <fstream>  
#include <new>  
#include <stack>  
#include <vector>  
#include <cstring>  
#include <cstdlib>  
#include <ctype>  
#include "mccommon.h"  
  
using namespace std;  
  
char *prog; // current execution point in source code  
char *p_buf; // points to start of program buffer  
  
// This structure encapsulates the info  
// associated with variables.  
struct var_type {  
    char var_name[MAX_ID_LEN+1]; // name  
    token_t v_type; // data type  
    int value; // value  
};  
  
// This vector holds info for global variables.  
vector<var_type> global_vars;
```

```

// This vector holds info for local variables
// and parameters.
vector<var_type> local_var_stack;

// This structure encapsulates function info.
struct func_type {
    char func_name[MAX_ID_LEN+1]; // name
    token_ireps ret_type; // return type
    char *loc; // location of entry point in program
};

// This vector holds info about functions.
vector<func_type> func_table;

// Stack for managing function scope.
stack<int> func_call_stack;

// Stack for managing nested scopes.
stack<int> nest_scope_stack;

char token[MAX_T_LEN+1]; // current token
tok_types token_type; // token type
token_ireps tok; // internal representation

int ret_value; // function return value

bool breakfound = false; // true if break encountered

int main(int argc, char *argv[])
{
    if(argc != 2) {
        cout << "Usage: minicpp <filename>\n";
        return 1;
    }

    // Allocate memory for the program.
    try {
        p_buf = new char[PROG_SIZE];
    } catch (bad_alloc exc) {
        cout << "Could Not Allocate Program Buffer\n";
        return 1;
    }

    // Load the program to execute.
    if(!load_program(p_buf, argv[1])) return 1;

    // Set program pointer to start of program buffer.
    prog = p_buf;

    try {
        // Find the location of all functions

```

```

// and global variables in the program.
prescan();

// Next, set up the call to main().

// Find program starting point.
prog = find_func("main");

// Check for incorrect or missing main() function.
if(!prog) {
    cout << "main() Not Found\n";
    return 1;
}

// Back up to opening (.
prog--;

// Set the first token to main
strcpy(token, "main");

// Call main() to start interpreting.
call();
}
catch(InterpExc exc) {
    sntx_err(exc.get_err());
    return 1;
}
catch(bad_alloc exc) {
    cout << "Out Of Memory\n";
    return 1;
}

return ret_value;
}

// Load a program.
bool load_program(char *p, char *fname)
{
    int i=0;

    ifstream in(fname, ios::in | ios::binary);
    if(!in) {
        cout << "Cannot Open file.\n";
        return false;
    }

    do {
        *p = in.get();
        p++; i++;
    } while(!in.eof() && i < PROG_SIZE);
    if(i == PROG_SIZE) {
        cout << "Program Too Big\n";
    }
}

```



```

    return false;
}

// Null terminate the program. Skip any EOF
// mark if present in the file.
if(*(p-2) == 0x1a) *(p-2) = '\0';
else *(p-1) = '\0';

in.close();

return true;
}

// Find the location of all functions in the program
// and store global variables.
void prescan()
{
    char *p, *tp;
    char temp[MAX_ID_LEN+1];
    token_ireps datatype;
    func_type ft;

    // When brace is 0, the current source position
    // is outside of any function.
    int brace = 0;

    p = prog;

    do {
        // Bypass code inside functions
        while(brace) {
            get_token();
            if(tok == END) throw InterpExc(UNBAL_BRACES);
            if(*token == '(') brace++;
            if(*token == ')') brace--;
        }

        tp = prog; // save current position
        get_token();

        // See if global var type or function return type.
        if(tok==CHAR || tok==INT) {
            datatype = tok; // save data type
            get_token();

            if(token_type == IDENTIFIER) {
                strcpy(temp, token);
                get_token();

                if(*token != '(') { // must be global var
                    prog = tp; // return to start of declaration
                    decl_global();
                }
            }
        }
    } while(tok != END);
}

```



```

    eval_exp(value); // process the expression
    if(*token != ';') throw InterpExc(SEMI_EXPECTED);
}
else if(token_type==BLOCK) { // block delimiter?
    if(*token == '{}') { // is a block
        block = 1; // interpreting block, not statement
        // Record nested scope.
        nest_scope_stack.push(local_var_stack.size());
    }
    else { // is a }, so reset scope and return
        // Reset nested scope.
        local_var_stack.resize(nest_scope_stack.top());
        nest_scope_stack.pop();
        return;
    }
}
else // is keyword
    switch(tok) {
        case CHAR:
        case INT: // declare local variables
            putback();
            decl_local();
            break;
        case RETURN: // return from function call
            func_ret();
            return;
        case IF: // process an if statement
            exec_if();
            break;
        case ELSE: // process an else statement
            find_eob(); // find end of else block
                        // and continue execution
            break;
        case WHILE: // process a while loop
            exec_while();
            break;
        case DO: // process a do-while loop
            exec_do();
            break;
        case FOR: // process a for loop
            exec_for();
            break;
        case BREAK: // handle break
            breakfound = true;

            // Reset nested scope.
            local_var_stack.resize(nest_scope_stack.top());
            nest_scope_stack.pop();
            return;
        case SWITCH: // handle a switch statement
            exec_switch();
            break;
    }

```

```

        case COUT: // handle console output
            exec_cout();
            break;
        case CIN: // handle console input
            exec_cin();
            break;
        case END:
            exit(0);
    }
} while (tok != END && block);
return;
}

// Return the entry point of the specified function.
// Return NULL if not found.
char *find_func(char *name)
{
    unsigned i;

    for(i=0; i < func_table.size(); i++)
        if(!strcmp(name, func_table[i].func_name))
            return func_table[i].loc;
    return NULL;
}

// Declare a global variable.
void decl_global()
{
    token_ireps vartype;
    var_type vt;

    get_token(); // get type

    vartype = tok; // save var type

    // Process comma-separated list.
    do {
        vt.v_type = vartype;
        vt.value = 0; // init to 0
        get_token(); // get name

        // See if variable is a duplicate.
        for(unsigned i=0; i < global_vars.size(); i++)
            if(!strcmp(global_vars[i].var_name, token))
                throw InterpExc(DUP_VAR);

        strcpy(vt.var_name, token);
        global_vars.push_back(vt);

        get_token();
    } while(*token == ',');
}

```

```

    if(*token != ';') throw InterpExc(SEMI_EXPECTED);
}

// Declare a local variable.
void decl_local()
{
    var_type vt;

    get_token(); // get var type
    vt.v_type = tok; // store type

    vt.value = 0; // init var to 0

    // Process comma-separated list.
    do {
        get_token(); // get var name

        // See if variable is already the name
        // of a local variable in this scope.
        if(!local_var_stack.empty())
            for(int i=local_var_stack.size()-1;
                i >= nest_scope_stack.top(); i--)
            {
                if(!strcmp(local_var_stack[i].var_name, token))
                    throw InterpExc(DUP_VAR);
            }

        strcpy(vt.var_name, token);
        local_var_stack.push_back(vt);
        get_token();
    } while(*token == ',');

    if(*token != ';') throw InterpExc(SEMI_EXPECTED);
}

// Call a function.
void call()
{
    char *loc, *temp;
    int lvartemp;

    // First, find entry point of function.
    loc = find_func(token);

    if(loc == NULL)
        throw InterpExc(FUNC_UNDEF); // function not defined
    else {
        // Save local var stack index.
        lvartemp = local_var_stack.size();

        get_args(); // get function arguments
        temp = prog; // save return location
    }
}

```

```

func_call_stack.push(lvartemp); // push local var index

prog = loc; // reset prog to start of function
get_params(); // load the function's parameters with
               // the values of the arguments

interp(); // interpret the function
prog = temp; // reset the program pointer

if(func_call_stack.empty()) throw InterpExc(RET_NOCALL);

// Reset local_var_stack to its previous state.
local_var_stack.resize(func_call_stack.top());
func_call_stack.pop();
}
}

// Push the arguments to a function onto the local
// variable stack.
void get_args()
{
    int value, count, temp[NUM_PARAMS];
    var_type vt;

    count = 0;
    get_token();
    if(*token != '(') throw InterpExc(PAREN_EXPECTED);

    // Process a comma-separated list of values.
    do {
        eval_exp(value);
        temp[count] = value; // save temporarily
        get_token();
        count++;
    } while(*token == ',');
    count--;

    // Now, push on local_var_stack in reverse order.
    for(; count>=0; count--) {
        vt.value = temp[count];
        vt.v_type = ARG;
        local_var_stack.push_back(vt);
    }
}

// Get function parameters.
void get_params()
{
    var_type *p;
    int i;

```

```

i = local_var_stack.size()-1;

// Process comma-separated list of parameters.
do {
    get_token();
    p = &local_var_stack[i];
    if(*token != ',') {
        if(tok != INT && tok != CHAR)
            throw InterpExc(TYPE_EXPECTED);

        p->v_type = tok;
        get_token();

        // Link parameter name with argument already on
        // local var stack.
        strcpy(p->var_name, token);
        get_token();
        i--;
    }
    else break;
} while(*token == ',');

if(*token != ')') throw InterpExc(PAREN_EXPECTED);
}

// Return from a function.
void func_ret()
{
    int value;

    value = 0;

    // Get return value, if any.
    eval_exp(value);

    ret_value = value;
}

// Assign a value to a variable.
void assign_var(char *vname, int value)
{
    // First, see if it's a local variable.
    if(!local_var_stack.empty())
        for(int i=local_var_stack.size()-1;
            i >= func_call_stack.top(); i--)
        {
            if(!strcmp(local_var_stack[i].var_name,
                vname))
            {
                if(local_var_stack[i].v_type == CHAR)
                    local_var_stack[i].value = (char) value;
                else if(local_var_stack[i].v_type == INT)

```

```

        local_var_stack[i].value = value;
    return;
}

// Otherwise, try global vars.
for(unsigned i=0; i < global_vars.size(); i++)
    if(!strcmp(global_vars[i].var_name, vname)) {
        if(global_vars[i].v_type == CHAR)
            global_vars[i].value = (char) value;
        else if(global_vars[i].v_type == INT)
            global_vars[i].value = value;
        return;
    }

throw InterpExc(NOT_VAR); // variable not found
}

// Find the value of a variable.
int find_var(char *vname)
{
    // First, see if it's a local variable.
    if(!local_var_stack.empty())
        for(int i=local_var_stack.size()-1;
            i >= func_call_stack.top(); i--)
        {
            if(!strcmp(local_var_stack[i].var_name, vname))
                return local_var_stack[i].value;
        }

    // Otherwise, try global vars.
    for(unsigned i=0; i < global_vars.size(); i++)
        if(!strcmp(global_vars[i].var_name, vname))
            return global_vars[i].value;

    throw InterpExc(NOT_VAR); // variable not found
}

// Execute an if statement.
void exec_if()
{
    int cond;

    eval_exp(cond); // get if expression.

    if(cond) { // if true, process target of IF
        // Confirm start of block.
        if(*token != '{')
            throw InterpExc(BRACE_EXPECTED);

        interp();
    }
}

```



```

else {
    // Otherwise skip around IF block and
    // process the ELSE, if present.

    find_eob(); // find start of next line
    get_token();

    if(tok != ELSE) {
        // Restore token if no ELSE is present.
        putback();
        return;
    }

    // Confirm start of block.
    get_token();
    if(*token != '{')
        throw InterpExc(BRACE_EXPECTED);
    putback();

    interp();
}

// Execute a switch statement.
void exec_switch()
{
    int sval, oval;
    int brace;

    eval_exp(sval); // Get switch expression.

    // Check for start of block.
    if(*token != '{')
        throw InterpExc(BRACE_EXPECTED);

    // Record new scope.
    nest_scope_stack.push(local_var_stack.size());

    // Now, check case statements.
    for(;;) {
        brace = 1;
        // Find a case statement.
        do {
            get_token();
            if(*token == '{') brace++;
            else if(*token == '}') brace--;
        } while(tok != CASE && tok != END && brace);

        // If no matching case found, then skip.
        if(!brace) break;

        if(tok == END) throw InterpExc(SYNTAX);
    }
}

```

```

// Get value of the case statement.
eval_exp(cval);

// Read and discard the :
get_token();

if(*token != ':')
    throw InterpExc(COLON_EXPECTED);

// If values match, then interpret.
if(cval == sval) {
    brace = 1;
    do {
        interp();

        if(*token == '{') brace++;
        else if(*token == '}') brace--;
    } while(!breakfound && tok != END && brace);

    // Find end of switch statement.
    while(brace) {
        get_token();
        if(*token == '{') brace++;
        else if(*token == '}') brace--;
    }
    breakfound = false;

    break;
}
}

// Execute a while loop.
void exec_while()
{
    int cond;
    char *temp;

    putback(); // put back the while
    temp = prog; // save location of top of while loop

    get_token();
    eval_exp(cond); // check the conditional expression

    // Confirm start of block.
    if(*token != '{')
        throw InterpExc(BRACE_EXPECTED);

    if(cond)
        interp(); // if true, interpret
    else { // otherwise, skip to end of loop

```

```

    find_eob();
    return;
}

prog = temp; // loop back to top

// Check for break in loop.
if(breakfound) {
    // Find start of loop block.
    do {
        get_token();
    } while(*token != '{' && tok != END );

    putback();
    breakfound = false;
    find_eob(); // now, find end of loop
    return;
}
}

// Execute a do loop.
void exec_do()
{
    int cond;
    char *temp;

    // Save location of top of do loop.
    putback(); // put back do
    temp = prog;

    get_token(); // get start of loop block

    // Confirm start of block.
    get_token();
    if(*token != '{')
        throw InterpExc(BRACE_EXPECTED);
    putback();

    interp(); // interpret loop

    // Check for break in loop.
    if(breakfound) {
        prog = temp;
        // Find start of loop block.
        do {
            get_token();
        } while(*token != '{' && tok != END);

        // Find end of while block.
        putback();
        find_eob();
    }
}

```

```

    // Now, find end of while expression.
    do {
        get_token();
    } while(*token != ';' && tok != END);
    if(tok == END) throw InterpExc(SYNTAX);

    breakfound = false;
    return;
}

get_token();
if(tok != WHILE) throw InterpExc(WHILE_EXPECTED);
eval_exp(cond); // check the loop condition

// If true loop; otherwise, continue on.
if(cond) prog = temp;
}

// Execute a for loop.
void exec_for()
{
    int cond;
    char *temp, *temp2;
    int paren ;

    get_token(); // skip opening (
    eval_exp(cond); // initialization expression

    if(*token != ';') throw InterpExc(SEMI_EXPECTED);
    prog++; // get past the ;
    temp = prog;

    for(;;) {
        // Get the value of the conditional expression.
        eval_exp(cond);

        if(*token != ';') throw InterpExc(SEMI_EXPECTED);
        prog++; // get past the ;
        temp2 = prog;

        // Find start of for block.
        paren = 1;
        while(paren) {
            get_token();
            if(*token == '(') paren++;
            if(*token == ')') paren--;
        }

        // Confirm start of block.
        get_token();
        if(*token != '{')

```

```

        throw InterpExc(BRACE_EXPECTED);
    putback();

    // If condition is true, interpret
    if(cond)
        interp();
    else { // otherwise, skip to end of loop
        find_eob();
        return;
    }

    prog = temp2; // go to increment expression

    // Check for break in loop.
    if(breakfound) {
        // Find start of loop block.
        do {
            get_token();
        } while(*token != '{' && tok != END);

        putback();
        breakfound = false;
        find_eob(); // now, find end of loop
        return;
    }

    // Evaluate the increment expression.
    eval_exp(cond);

    prog = temp; // loop back to top
}

// Execute a cout statement.
void exec_cout()
{
    int val;

    get_token();
    if(*token != LS) throw InterpExc(SYNTAX);

    do {
        get_token();

        if(token_type==STRING) {
            // Output a string.
            cout << token;
        }
        else if(token_type == NUMBER ||
                token_type == IDENTIFIER) {
            // Output a number.
            putback();

```

```

        eval_exp(val);
        cout << val;
    }
    else if(*token == '\\') {
        // Output a character constant.
        putback();
        eval_exp(val);
        cout << (char) val;
    }

    get_token();
} while(*token == LS);

if(*token != ';') throw InterpExc(SEMI_EXPECTED);
}

// Execute a cin statement.
void exec_cin()
{
    int val;
    char chval;
    token_ireps vtype;

    get_token();
    if(*token != RS) throw InterpExc(SYNTAX);

    do {
        get_token();
        if(token_type != IDENTIFIER)
            throw InterpExc(NOT_VAR);

        vtype = find_var_type(token);

        if(vtype == CHAR) {
            cin >> chval;
            assign_var(token, chval);
        }
        else if(vtype == INT) {
            cin >> val;
            assign_var(token, val);
        }

        get_token();
    } while(*token == RS);
    if(*token != ';') throw InterpExc(SEMI_EXPECTED);
}

// Find the end of a block.
void find_eob()
{
    int brace;

```

```

    get_token();
    if(*token != '{' )
        throw InterpExc(BRACE_EXPECTED);

    brace = 1;

    do {
        get_token();
        if(*token == '{') brace++;
        else if(*token == '}') brace--;
    } while(brace && tok != END);

    if(tok==END) throw InterpExc(UNBAL_BRACES);
}

// Determine if an identifier is a variable. Return
// true if variable is found; false otherwise.
bool is_var(char *vname)
{
    // See if vname is a local variable.
    if(!local_var_stack.empty())
        for(int i=local_var_stack.size()-1;
            i >= func_call_stack.top(); i--)
        {
            if(!strcmp(local_var_stack[i].var_name, vname))
                return true;
        }

    // See if vname is a global variable.
    for(unsigned i=0; i < global_vars.size(); i++)
        if(!strcmp(global_vars[i].var_name, vname))
            return true;

    return false;
}

// Return the type of variable.
token_ireps find_var_type(char *vname)
{
    // First, see if it's a local variable.
    if(!local_var_stack.empty())
        for(int i=local_var_stack.size()-1;
            i >= func_call_stack.top(); i--)
        {
            if(!strcmp(local_var_stack[i].var_name, vname))
                return local_var_stack[i].v_type;
        }

    // Otherwise, try global vars.
    for(unsigned i=0; i < global_vars.size(); i++)
        if(!strcmp(global_vars[i].var_name, vname))
            return local_var_stack[i].v_type;
}

```

```

    return UNDEFTOK;
}

```

9.6.1 main()函数

main()函数开始解释您在命令行中指定的程序。如下所示:

```

int main(int argc, char *argv[])
{
    if(argc != 2) {
        cout << "Usage: minicpp <filename>\n";
        return 1;
    }

    // Allocate memory for the program.
    try {
        p_buf = new char[PROG_SIZE];
    } catch (bad_alloc exc) {
        cout << "Could Not Allocate Program Buffer\n";
        return 1;
    }

    // Load the program to execute.
    if(!load_program(p_buf, argv[1])) return 1;

    // Set program pointer to start of program buffer.
    prog = p_buf;
    try {
        // Find the location of all functions
        // and global variables in the program.
        prescan();

        // Next, set up the call to main().

        // Find program starting point.
        prog = find_func("main");

        // Check for incorrect or missing main() function.
        if(!prog) {
            cout << "main() Not Found\n";
            return 1;
        }

        // Back up to opening (.
        prog--;

        // Set the first token to main
        strcpy(token, "main");

        // Call main() to start interpreting.
        call();
    }
}

```



```

    }

    catch(InterpExc exc) {
        sntx_err(exc.get_err());
        return 1;
    }
    catch(bad_alloc exc) {
        cout << "Out Of Memory\n";
        return 1;
    }

    return ret_value;
}

```

`main()`函数开始分配内存以保存被解释的程序。注意，可以被解释的最大程序由常量 `PROG_SIZE` 指定。这个值被任意地设置为 10000，但是如果您愿意，可以增加它。随后，调用函数 `load_program()` 函数加载这个程序。在这个程序被加载之后，`main()`函数执行了 3 个主要的动作：

- (1) 调用解释程序的预扫描函数 `prescan()`。
- (2) 查找 `main()` 在程序中的位置，使得解释程序准备好调用 `main()`。
- (3) 执行 `call()` 函数，在 `main()` 的起始位置开始执行这个程序。

`main()`函数还处理由 Mini C++ 生成的 `InterpExc` 异常。这包括解析器抛出的那些异常。

下面部分详细讨论解释程序的关键组成部分。

9.6.2 解释程序的预扫描程序

解释程序在开始执行程序之前，必须执行两个重要的记录任务：

- (1) 必须找到并初始化所有的全局变量。
- (2) 必须发现程序中每个函数定义的位置。

这两个任务由解释程序的预扫描程序完成。

在 Mini C++ 中，所有可执行的代码都存在于函数内部，因此一旦开始执行，Mini C++ 的解释程序没有理由到函数的外面。然而，全局变量的声明语句在所有函数的外面。因此，有必要使用预扫描程序来处理这些声明。解释程序没有其他(有效的)方法来了解它们。

为了保证执行的速度，需要知道每个函数在程序中定义的位置(尽管在技术上并不需要)，从而加快函数的调用。如果没有执行这个步骤，就需要搜索冗长的源代码，以找到每次调用函数的入口点。

寻找所有函数的入口点还有另一个目的。您知道，C++ 程序不是从这个程序的顶部开始执行的。相反，它是从 `main()` 函数开始的。另外，并没有要求 `main()` 函数是程序的第一个函数。从而有必要在程序的源代码中找到 `main()` 函数的位置，因此可以在那个点开始执行。(全局变量可能在 `main()` 函数的前面，因此，即使 `main()` 是第一个函数，也不一定是第一行代码)。由于预扫描程序找到了每个函数的入口点，它也就找到了 `main()` 函数的入口点。

执行预扫描的函数是 `prescan()`。如下所示：

```

// Find the location of all functions in the program
// and store global variables.

```

```

void prescan()
{
    char *p, *tp;
    char temp[MAX_ID_LEN+1];
    token_ireps datatype;
    func_type ft;

    // When brace is 0, the current source position
    // is outside of any function.
    int brace = 0;

    p = prog;

    do {
        // Bypass code inside functions
        while(brace) {
            get_token();
            if(tok == END) throw InterpExc(UNBAL_BRACES);
            if(*token == '(') brace++;
            if(*token == ')') brace--;
        }

        tp = prog; // save current position
        get_token();

        // See if global var type or function return type.
        if(tok==CHAR || tok==INT) {
            datatype = tok; // save data type
            get_token();

            if(token_type == IDENTIFIER) {
                strcpy(temp, token);
                get_token();

                if(*token != '(') { // must be global var
                    prog = tp; // return to start of declaration
                    decl_global();
                }
                else if(*token == '(') { // must be a function

                    // See if function already defined.
                    for(unsigned i=0; i < func_table.size(); i++)
                        if(!strcmp(func_table[i].func_name, temp))
                            throw InterpExc(DUP_FUNC);

                    ft.loc = prog;
                    ft.ret_type = datatype;
                    strcpy(ft.func_name, temp);
                    func_table.push_back(ft);

                    do {
                        get_token();

```

```

        } while(*token != ' ');
        // Next token will now be opening curly
        // brace of function.
    }
    else putback();
}
}
else {
    if(*token == '{') brace++;
    if(*token == '}') brace--;
}
} while(tok != END);
if(brace) throw InterpExc(UNBAL_BRACES);
prog = p;
}

```

prescan()函数按如下的方式运行。每次遇到左花括号时, brace 增 1。每次碰到右花括号时, brace 减 1。因此, 当 brace 大于 0 时, 从函数中读取当前的令牌。然而, 如果找到变量时, 则 brace 等于 0, 预扫描程序就知道这一定是一个全局变量。以相同的方式, 如果 brace 等于 0 时遇到函数名, 那么这一定是函数的定义(记住, Mini C++ 不支持函数原型)。

将全局变量存储在名为 global_vars 的向量中, 这个向量具有如下所示的 var_type 类型的结构:

```

// This structure encapsulates the info
// associated with variables.
struct var_type {
    char var_name[MAX_ID_LEN+1]; // name
    token_ireps v_type; // data type
    int value; // value
};

```

这个结构存储了变量的名称、值以及类型。

全局变量由如下所示的 decl_global()函数存储到 global_vars 向量中:

```

// Declare a global variable.
void decl_global()
{
    token_ireps vartype;
    var_type vt;

    get_token(); // get type

    vartype = tok; // save var type

    // Process comma-separated list.
    do {
        vt.v_type = vartype;
        vt.value = 0; // init to 0
        get_token(); // get name

        // See if variable is a duplicate.
    } while (true);
}

```

```

    for(unsigned i=0; i < global_vars.size(); i++)
        if(!strcmp(global_vars[i].var_name, token))
            throw InterpExc(DUP_VAR);

    strcpy(vt.var_name, token);
    global_vars.push_back(vt);

    get_token();
} while(*token == ',');

if(*token != ';') throw InterpExc(SEMI_EXPECTED);
}

```

本质上，`decl_global()`函数获取变量的类型和名称，将其初始化为 0，然后放到 `global_vars` 的结尾。然而，首先要执行一个检查，以确保没有已经声明具有相同名称的变量。

每个用户自定义函数的位置被放入名为 `func_table` 的向量中，这个向量存储 `func_type` 类型的结构，如下所示：

```

// This structure encapsulates function info.
struct func_type {
    char func_name[MAX_ID_LEN+1]; // name
    token_ireps ret_type; // return type
    char *loc; // location of entry point in program
};

```

每个函数的入口都包含了返回类型、名称以及函数入口点在源代码中的位置。在函数存储到这个表格之前，`prescan()`函数检查是否存在相同名称的函数。注意没有存储参数信息。这个信息在运行时实际调用函数时获取。

9.6.3 `interp()`函数

`interp()`函数是解释程序的核心。正是这个函数基于输入流中的下一个令牌决定了采取的操作。这个函数用来解释一个代码单元并返回。如果这个单元由一条语句组成，那么这条语句被解释，`interp()`函数返回。然而，如果读取了一个左花括号，那么这个代码块中的所有语句都会被解释。这是由 `block` 标志管理的，如果读取了左花括号，则将 `block` 设置为 1。`interp()`函数一直解释后续的语句，直到读取右花括号。`interp()`函数如下所示：

```

// Interpret a single statement or block of code. When
// interp() returns from its initial call, the final
// brace (or a return) in main() has been encountered.
void interp()
{
    int value;
    int block = 0;

    do {
        // Don't interpret until break is handled.
        if(breakfound) return;

        token_type = get_token();
    }
}

```

```

// See what kind of token is up.
if(token_type == IDENTIFIER ||
    *token == INC || *token == DEC)
{
    // Not a keyword, so process expression.
    putback(); // restore token to input stream for
               // further processing by eval_exp()
    eval_exp(value); // process the expression
    if(*token != ';' ) throw InterpExc(SEMI_EXPECTED);
}
else if(token_type==BLOCK) { // block delimiter?
    if(*token == '{') { // is a block
        block = 1; // interpreting block, not statement
        // Record nested scope.
        nest_scope_stack.push(local_var_stack.size());
    }
    else { // is a }, so reset scope and return
        // Reset nested scope.
        local_var_stack.resize(nest_scope_stack.top());
        nest_scope_stack.pop();
        return;
    }
}
else // is keyword
    switch(tok) {
        case CHAR:
        case INT: // declare local variables
            putback();
            decl_local();
            break;
        case RETURN: // return from function call
            func_ret();
            return;
        case IF: // process an if statement
            exec_if();
            break;
        case ELSE: // process an else statement
            find_eob(); // find end of else block
                       // and continue execution
            break;
        case WHILE: // process a while loop
            exec_while();
            break;
        case DO: // process a do-while loop
            exec_do();
            break;
        case FOR: // process a for loop
            exec_for();
            break;
        case BREAK: // handle break
            breakfound = true;
    }
}

```

```

        // Reset nested scope.
        local_var_stack.resize(nest_scope_stack.top());
        nest_scope_stack.pop();
        return;
    case SWITCH: // handle a switch statement
        exec_switch();
        break;
    case COUT: // handle console output
        exec_cout();
        break;
    case CIN: // handle console input
        exec_cin();
        break;
    case END:
        exit(0);
}
} while (tok != END && block);
return;
}

```

除了调用类似于 `exit()` 的函数之外，C++ 程序在遇到 `main()` 中的最后一个花括号(或者 `return`)时结束，——而不一定在源代码的最后一行结束。这也是 `interp()` 只执行一条语句或者一个代码块而不是整个程序的原因之一。另外从概念上讲，C++ 由代码块组成。因此，每次遇到新代码块时，都会调用 `interp()` 函数。这包括函数块以及由各种 C++ 语句开始的代码块，如 `if`。在执行程序的过程中，Mini C++ 有可能递归调用 `interp()` 函数。

`interp()` 函数的运行方式如下。首先，检查程序中是否有 `break` 语句。如果有一条，则全局变量 `breakfound` 为 `true`。这个变量一直保持到被解释程序的其他部分清除，当描述控制语句的解释时，您将会看到这一点。

假定 `breakfound` 为 `false`，则 `interp()` 函数从程序中读取下一个令牌。如果这个令牌是标识符，则这条语句一定是一个表达式，从而会调用表达式解析器。由于表达式解析器希望读取表达式本身的第一个令牌，通过调用 `putback()` 函数，这个令牌返回到输入流。当 `eval_exp()` 返回时，`token` 将保存表达式解析器读取的最后一个令牌。如果 `token` 没有包含分号，就会报告错误。

如果从程序中读取的下一个令牌是左花括号，那么 `block` 设置为 1，`local_var_stack` 当前的大小被压入 `nest_scope_stack`。`local_var_stack` 用于保存所有的局部变量，包括在嵌套的块作用域中声明的变量。相反，如果下一个令牌是右花括号，则 `local_var_stack` 的大小缩短为 `nest_scope_stack` 顶部所指定的大小。从而有效地删除了这个作用域内声明的任何局部变量。因此，“{”会导致保存局部变量堆栈的大小，“}”导致局部变量堆栈重新设置为其原始大小。这个机制支持嵌套的局部作用域。

最后，如果这个令牌是一个关键字，则会执行 `switch` 语句，并调用合适的例程来处理这条语句。`get_token()` 给定关键字的等价整型值的原因是为了能够使用 `switch` 语句，而不是使用涉及到字符串比较的一系列 `if` 语句(这样做相当慢)。

9.6.4 处理局部变量

当解释程序遇到 `int` 或者 `char` 关键字时，会调用 `decl_local()` 函数为局部变量创建存储空间。

前面讲过，一旦程序开始执行，解释程序就不会遇到全局变量的声明语句，因为只执行函数中的代码。因此，如果发现了变量声明语句，那么一定是一个局部变量的声明(或者参数，在下一部分讨论)。通常，局部变量存储在堆栈中。如果对语言进行编译，则通常使用系统堆栈。然而，在解释模式中，解释程序必须维护局部变量的堆栈。

在 Mini C++ 中，`local_var_stack` 为局部变量提供了一个堆栈。有些出人意料的是，`local_var_stack` 是一个 `vector` 容器而不是 `stack` 容器。原因是：尽管以堆栈的风格来管理局部变量，然而在需要访问变量值时，必须能够按顺序搜索容器。`stack` 容器不能方便地提供这种顺序搜索，而 `vector` 容器可以。

`decl_local()` 函数如下所示：

```
// Declare a local variable.
void decl_local()
{
    var_type vt;

    get_token(); // get var type
    vt.v_type = tok; // store type

    vt.value = 0; // init var to 0

    // Process comma-separated list.
    do {
        get_token(); // get var name

        // See if variable is already the name
        // of a local variable in this scope.
        if(!local_var_stack.empty())
            for(int i=local_var_stack.size()-1;
                i >= nest_scope_stack.top(); i--)
            {
                if(!strcmp(local_var_stack[i].var_name, token))
                    throw InterpExc(DUP_VAR);
            }

        strcpy(vt.var_name, token);
        local_var_stack.push_back(vt);
        get_token();
    } while(*token == ',');

    if(*token != ';') throw InterpExc(SEMI_EXPECTED);
}
```

每次遇到局部变量时，都会将其名称、类型以及值(初始值为 0)压入堆栈。这个过程如下。`decl_local()` 函数首先读取要声明的变量的类型，并将其初始值设置为 0。随后，进入一个循环，在循环内读取由逗号分割的标识符的名称。每循环一次，关于每个变量的信息都会压入到局部变量堆栈中。在这个过程中，进行了一个检查，以确保当前的作用域内不存在相同名称的变量。(我们发现，在当前作用域内声明的变量在 `local_var_stack` 的当前栈顶以及保存在 `nest_scope_stack` 栈顶的索引之间)。最后，检查最后的令牌，以确保它包含分号。

9.6.5 调用用户自定义的函数

为 C++ 实现一个解释程序最困难的部分可能是管理用户自定义函数的执行。解释程序不仅要在新的位置读取源代码,然后在函数结束后返回到调用例程,解释程序还必须处理 3 个任务:参数的传递、参数的分配以及函数的返回值。

所有的函数调用(除了最初的对 main() 的调用)都是通过表达式解析器从 atom() 函数中由 call() 调用。实际上处理函数调用细节的是 call() 函数。call() 函数如下所示,让我们仔细分析一下:

```
// Call a function.
void call()
{
    char *loc, *temp;
    int lvartemp;

    // First, find entry point of function.
    loc = find_func(token);

    if(loc == NULL)
        throw InterpExc(FUNC_UNDEF); // function not defined
    else {
        // Save local var stack index.
        lvartemp = local_var_stack.size();

        get_args(); // get function arguments
        temp = prog; // save return location

        func_call_stack.push(lvartemp); // push local var index

        prog = loc; // reset prog to start of function
        get_params(); // load the function's parameters with
                     // the values of the arguments

        interp(); // interpret the function

        prog = temp; // reset the program pointer

        if(func_call_stack.empty()) throw InterpExc(RET_NOCALL);

        // Reset local_var_stack to its previous state.
        local_var_stack.resize(func_call_stack.top());
        func_call_stack.pop();
    }
}
```

call() 函数做的第一件事情是通过调用 find_func() 函数来寻找源代码中指定函数的入口点的位置。随后在 lvartemp 变量中保存了局部变量堆栈的当前大小。然后调用了 get_args() 来处理任何函数参数。get_args() 函数读取逗号分割的表达式列表,并且以相反的顺序将其压入局部变量堆栈中(以相反的顺序将表达式压入堆栈,当解释函数时,可以比较容易地与对应的参数匹配)。当这些值被压入时,没有给定名称。函数参数的名称是由 get_params() 函数给定的,稍后将讨

论这个函数。

一旦完成函数参数的处理, `prog` 的当前值就被保存在 `temp` 中。这个位置是函数的返回点。随后, `lvartemp` 的值被压入函数调用堆栈 `func_call_stack` 中。其目的是在每次调用函数时, 存储局部变量堆栈顶部的索引。这个值代表了与被调用函数相关的变量(以及参数)相关的局部变量堆栈的开始点。使用函数调用堆栈顶部的值阻止函数访问不是它声明的任何局部变量。

下面的两行代码将程序指针指向函数的开始, 并调用 `get_params()` 将函数形参的名称与保存在局部变量堆栈中的参数的值链接起来。为此, `get_params()` 读取每个参数, 并将其名称复制到 `local_var_stack` 中已经存储的对应的参数。

函数的实际执行是通过调用 `interp()` 完成的。当 `interp()` 返回时, 程序指针(`prog`)重新设置为返回点, 局部变量堆栈索引重新设置为函数调用前的值。最后一步有效地从堆栈中删除所有的函数局部变量和参数。

如果被调用的函数包含了 `return` 语句, 那么 `interp()` 会在返回到 `call()` 函数之前调用 `func_ret()`。这个函数处理任何返回值, 如下所示:

```
// Return from a function.
void func_ret()
{
    int value;

    value = 0;

    // Get return value, if any.
    eval_exp(value);

    ret_value = value;
}
```

全局变量 `ret_value` 是一个保存了函数返回值的整型数。一眼看上去, 您可能会有些迷惑, 为什么使用局部变量 `value` 而不是 `ret_value` 来接收 `eval_exp()` 返回的值。原因是函数可能是递归的, `eval_exp()` 可能需要调用相同的函数来获取它的值。在此情况下, 全局变量有可能被改写, 因此不能用来接收这个值。

9.6.6 给变量赋值

我们暂时返回到表达式解析器。当遇到赋值语句时, 计算表达式右边的值, 并调用 `assign_var()` 将这个值赋给左边的变量。然而, C++ 支持各种作用域, 包括全局作用域(正式的名称是命名空间作用域)和局部作用域。另外, 局部作用域可以是嵌套的。这很重要, 因为它影响到了 Mini C++ 寻找变量值的方式。为了理解这种方式, 考虑下面的 Mini C++ 程序:

```
int count;

int main()
{
    int count, i;

    count = 100;
```

```

    i = f();

    return 0;
}

int f()
{
    int count;

    count = 99;

    return count;
}

```

当赋予变量 `count` 一个值时, `assign_var()` 函数如何知道使用哪个变量呢? 是全局的 `count` 变量还是局部的 `count` 变量? 答案很简单。在 C++ 中, 局部变量的优先级高于相同名称的全局变量的优先级。另外, 局部变量在它的代码块之外不会被发现。为了观察如何使用这些规则来解决前面的赋值, 检查如下所示的 `assign_var()` 函数:

```

// Assign a value to a variable.
void assign_var(char *vname, int value)
{
    // First, see if it's a local variable.
    if(!local_var_stack.empty())
        for(int i=local_var_stack.size()-1;
            i >= func_call_stack.top(); i--)
        {
            if(!strcmp(local_var_stack[i].var_name,
                        vname))
            {
                if(local_var_stack[i].v_type == CHAR)
                    local_var_stack[i].value = (char) value;
                else if(local_var_stack[i].v_type == INT)
                    local_var_stack[i].value = value;
                return;
            }
        }

    // Otherwise, try global vars.
    for(unsigned i=0; i < global_vars.size(); i++)
        if(!strcmp(global_vars[i].var_name, vname)) {
            if(global_vars[i].v_type == CHAR)
                global_vars[i].value = (char) value;
            else if(global_vars[i].v_type == INT)
                global_vars[i].value = value;
            return;
        }

    throw InterpExc(NOT_VAR); // variable not found
}

```

正如前面部分所解释的那样，每次调用函数时，局部变量堆栈(local_var_stack)顶部的当前索引被压入函数调用堆栈(func_call_stack)。这意味着函数定义的任何局部变量(或者参数)在这个点上被压入堆栈。因此，assign_var()函数首先搜索 local_var_stack，从堆栈的当前栈顶开始，当索引到达最近一次的函数调用时停止。这个机制确保了只检查这个函数的局部变量。(另外这种机制还有助于支持递归函数，因为每次调用函数时，栈顶的当前值都会被保存)。因此，在这个示例中，main()函数中的这一行

```
count=100;
```

导致 assign_var()寻找 main()中的局部变量 count。在 f()中，语句

```
count=99;
```

导致 assign_var()寻找 f()中声明的 count。

如果没有局部变量与某个变量名称匹配，就会搜索全局变量的名称列表。如果局部变量和全局变量的列表都没有包含这个变量，就会发生语法错误。

现在考虑嵌套循环中声明的变量的情况，如下所示：

```
int f(int n)
{
    int count;

    count = 99

    if(n > 0) {
        int count; // this count is local to the if

        count = 100; // refers to count in if block
        // ...
    }

    return count; // refers to outer count.
}
```

在此情况下，外部变量 count(函数代码开始时声明的)首先被压入堆栈 local_var_stack。然后 if 代码块中的局部变量 count 被压入 local_var_stack。因此，当执行下面的语句时：

```
count=100;// refers to count in if block
```

assign_var()查找 count，并首先发现 if 代码块中的局部变量 count 的副本，正如它应当做的那样。

最后一点，在 interp()函数中，每当离开一个代码块时，local_var_stack 都会缩短到进入这个代码块之前的长度，从而有效地从堆栈中删除这个代码块中声明的所有局部变量。每次函数返回时，local_var_stack 也会缩短，从而删除了与函数相关的所有局部变量和参数。

9.6.7 执行 if 语句

既然已经讨论了 Mini C++解释程序的基本结构，现在是观察控制语句如何实现的时候了。通常，每次遇到关键字语句时，interp()就会调用处理这条语句的函数。解释各种控制语句的函

数都以前缀 `exec_` 开头。例如, `exec_for()` 函数解释 `for`, `exec_switch()` 函数解释 `switch` 等。

最容易解释的控制语句之一是 `if` 语句。`exec_if()` 函数处理 `if` 语句, 代码如下所示:

```
// Execute an if statement.
void exec_if()
{
    int cond;

    eval_exp(cond); // get if expression.

    if(cond) { // if true, process target of IF
        // Confirm start of block.
        if(*token != '{')
            throw InterpExc(BRACE_EXPECTED);

        interp();
    }
    else {
        // Otherwise skip around IF block and
        // process the ELSE, if present.

        find_eob(); // find start of next line
        get_token();

        if(tok != ELSE) {
            // Restore token if no ELSE is present.
            putback();
            return;
        }

        // Confirm start of block.
        get_token();
        if(*token != '{')
            throw InterpExc(BRACE_EXPECTED);
        putback();

        interp();
    }
}
```

让我们仔细分析这个函数的操作。

首先, `exec_if()` 调用 `val_exp()` 求条件表达式的值。如果条件为 `true` (非 0), 则这个函数调用 `interp()`, `interp()` 执行 `if` 代码块中的代码。如果条件为 `false`, 则调用 `find_eob()`, 这个函数立即将程序指针提前到 `if` 代码块结尾后的位置。如果存在 `else` 语句, 则执行与 `else` 相关的代码块。否则, 简单地开始下一行代码的执行。

如果执行 `if` 代码块, 并且存在 `else` 代码块, 则必须有某种方式来忽略掉 `else` 代码块。当遇到 `else` 语句时, `interp()` 完成了这个任务。在此情况下, `interp()` 简单地调用 `find_eob()` 来忽略掉 `else` 代码块。在此之后, 继续执行 `else` 代码块之后的第一条语句。记住, (在语法正确的程序中) 只有在执行 `if` 代码块之后, `interp()` 才会处理 `else`。

另外一点是：注意 `exec_if()` 将确认 `if` 和 `else` 的目标代码包含在一个代码块中。正如前面所述，为了简化解释程序，所有控制语句的目标必须是代码块。该限制使解释程序代码块连接成一个整体。

9.6.8 switch 语句和 break 语句

解释 `switch` 语句需要的工作多于解释 `if` 语句。原因之一是它的语法更加复杂。另一个原因是它与 `break` 语句有关。因此，为了处理 `switch` 语句，意味着必须处理 `break` 语句。在此对二者进行了分析。

处理 `break` 语句相当容易，因为无论在 `switch` 语句还是在循环中使用，它都执行相同的操作：退出与这条语句相关的代码块。Mini C++ 用名为 `breakfound` 的全局标记变量来处理 `break` 语句，其初始值为 `false`。当发现 `break` 语句时，`breakfound` 设置为 `true`。然后用 `switch` 语句(以及后面提到的循环语句)来检查这个变量，以判断是否已经执行 `break` 语句。如果 `breakfound` 为 `true`，则终止当前的代码块，并将 `breakfound` 重新设置为 `false`。

`break` 语句由如下所示的 `interp()` 中的代码处理：

```
case BREAK: // handle break
    breakfound = true;

    // Reset nested scope.
    local_var_stack.resize(nest_scope_stack.top());
    nest_scope_stack.pop();
    return;
```

正如您所看到的那样，除了将 `breakfound` 设置为 `true` 之外，局部变量堆栈被重新设置为被终止的代码块开始之前的状态。变量可以在任何代码块内声明，并局限于这个代码块。因此，当发现 `break` 时，与封闭的代码块相关的任何局部变量都必须被删除。

处理 `switch` 语句的 `exec_switch()` 函数如下所示：

```
// Execute a switch statement.
void exec_switch()
{
    int sval, cval;
    int brace;

    eval_exp(sval); // Get switch expression.

    // Check for start of block.
    if(*token != '{')
        throw InterpExc(BRACE_EXPECTED);

    // Record new scope.
    nest_scope_stack.push(local_var_stack.size());

    // Now, check case statements.
    for(;;) {
        brace = 1;
        // Find a case statement.
```

```

do {
    get_token();
    if(*token == '{') brace++;
    else if(*token == '}') brace--;
} while(tok != CASE && tok != END && brace);

// If no matching case found, then skip.
if(!brace) break;

if(tok == END) throw InterpExc(SYNTAX);

// Get value of the case statement.
eval_exp(cval);

// Read and discard the :
get_token();

if(*token != ':')
    throw InterpExc(COLON_EXPECTED);

// If values match, then interpret.
if(cval == sval) {
    brace = 1;
    do {
        interp();

        if(*token == '{') brace++;
        else if(*token == '}') brace--;
    } while(!breakfound && tok != END && brace);

    // Find end of switch statement.
    while(brace) {
        get_token();
        if(*token == '{') brace++;
        else if(*token == '}') brace--;
    }
    breakfound = false;
    break;
}
}
}

```

首先, `exec_switch()` 函数获取 `switch` 表达式的值, 并将这个值存储在 `sval` 中。随后, 它检查 `switch` 代码块的开始, 并将局部变量堆栈的栈顶存储在 `nest_scope_stack` 中。这个步骤是必须的, 因为 `switch` 语句创建了嵌套的作用域。随后, 检查每个 `case` 语句的值, 直到遇到与 `sval` 中的值匹配的 `case` 或者到达 `switch` 语句的结尾。(为了简单起见, Mini C++ 没有支持 `default` 语句)。如果找到了匹配 `case`, 则执行与这个 `case` 相关的语句, 直到遇到 `break` 语句(也就是说, 直到 `breakfound` 为 `true`), 或者直到发现了 `switch` 代码块的结尾。在遇到 `break` 语句之后, 找到 `switch` 代码块的结尾, 则 `exec_switch()` 函数结束, 然后将 `breakfound` 设置为 `false`。

9.6.9 处理 while 循环

解释 while 循环相当容易。执行这个任务的函数是 `exec_while()`，代码如下所示：

```
// Execute a while loop.
void exec_while()
{
    int cond;
    char *temp;

    putback(); // put back the while
    temp = prog; // save location of top of while loop

    get_token();
    eval_exp(cond); // check the conditional expression

    // Confirm start of block.
    if(*token != '{')
        throw InterpExc(BRACE_EXPECTED);

    if(cond)
        interp(); // if true, interpret
    else { // otherwise, skip to end of loop
        find_eob();
        return;
    }

    prog = temp; // loop back to top

    // Check for break in loop.
    if(breakfound) {
        // Find start of loop block.
        do {
            get_token();
        } while(*token != '{' && tok != END);

        putback();
        breakfound = false;
        find_eob();
        return;
    }
}
```

`exec_while()`函数运行方式如下。首先，`while` 令牌被放回到输入流，`while` 在程序中的位置被保存在 `temp` 指针中。这个地址用来允许解释程序循环回 `while` 的顶部。随后，为了从输入流中删除 `while`，它被重新读取，并调用 `eval_exp()`来求 `while` 的条件表达式的值。如果条件表达式为 `true`，则调用 `interp()`来解释 `while` 代码块。当 `interp()`返回时，`prog`(程序指针)被设置为 `while` 循环开始的位置，这样做导致当控制返回到 `interp()`时，程序执行在循环的顶部重新启动，从而导致了循环的下一一次迭代。然而，如果 `interp()`是因为在循环内发现了 `break` 语句而返回，迭代会终止，找到 `while` 代码块的结尾，`exec_while()`函数返回。当条件表达式为 `false` 时，`while` 代

码块的结尾被发现，函数返回。

9.6.10 处理 do-while 循环

do-while 循环的处理方式非常类似于 while。当 interp() 遇到 do 语句的时候，调用 exec_do() 函数，代码如下所示：

```
// Execute a do loop.
void exec_do()
{
    int cond;
    char *temp;

    // Save location of top of do loop.
    putback(); // put back do
    temp = prog;

    get_token(); // get start of loop block

    // Confirm start of block.
    get_token();
    if(*token != '{')
        throw InterpExc(BRACE_EXPECTED);
    putback();

    interp(); // interpret loop

    // Check for break in loop.
    if(breakfound) {
        prog = temp;
        // Find start of loop block.
        do {
            get_token();
        } while(*token != '{' && tok != END);

        // Find end of while block.
        putback();
        find_eob();

        // Now, find end of while expression.
        do {
            get_token();
        } while(*token != ';' && tok != END);
        if(tok == END) throw InterpExc(SYNTAX);

        breakfound = false;
        return;
    }

    get_token();
    if(tok != WHILE) throw InterpExc(WHILE_EXPECTED);
```



```

eval_exp(cond); // check the loop condition

// If true loop; otherwise, continue on.
if(cond) prog = temp;
}

```

do-while 循环和 while 循环之间的主要区别是，do-while 至少会执行一次它的代码块，因为条件表达式在循环的底部。因此，`exec_do()` 首先将循环顶部的位置保存在 `temp` 中，然后调用 `interp()` 来解释与这个循环相关的代码块。当 `interp()` 返回时，获取对应的 `while`，并且求条件表达式的值。如果条件表达式为 `true`，`prog` 被重新设置为循环的顶部；否则，继续向下执行。如果遇到 `break` 语句，迭代终止，找到循环代码块的结尾。

9.6.11 for 循环

相对于其他循环而言，对 `for` 循环的解释是一个更加艰难的挑战。部分原因是 C++ 语言 `for` 循环的结构设计考虑了编译。主要问题是必须在每个循环的顶部检查 `for` 循环的条件表达式，而增量部分发生在循环的底部。因此，虽然 `for` 循环的这两个部分在源代码中连续出现，但它们的解释却被迭代的代码块分开。然而，多花一点力气，`for` 循环还是可以被正确地解释的。

当 `interp()` 遇到 `for` 语句时，会调用 `exec_for()` 函数。函数代码如下所示：

```

// Execute a for loop.
void exec_for()
{
    int cond;
    char *temp, *temp2;
    int paren ;

    get_token(); // skip opening (
    eval_exp(cond); // initialization expression

    if(*token != ';') throw InterpExc(SEMI_EXPECTED);
    prog++; // get past the ;
    temp = prog;

    for(;;) {
        // Get the value of the conditional expression.
        eval_exp(cond);

        if(*token != ';') throw InterpExc(SEMI_EXPECTED);
        prog++; // get past the ;
        temp2 = prog;

        // Find start of for block.
        paren = 1;
        while(paren) {
            get_token();
            if(*token == '(') paren++;
            if(*token == ')') paren--;
        }
    }
}

```

```

// Confirm start of block.
get_token();
if(*token != '{')
    throw InterpExc(BRACE_EXPECTED);
putback();

// If condition is true, interpret
if(cond)
    interp();
else { // otherwise, skip to end of loop
    find_eob();
    return;
}

prog = temp2; // go to increment expression

// Check for break in loop.
if(breakfound) {
    // Find start of loop block.
    do {
        get_token();
    } while(*token != '{' && tok != END);

    putback();
    breakfound = false;
    find_eob();
    return;
}

// Evaluate the increment expression.
eval_exp(cond);

prog = temp; // loop back to top
}
}

```

这个函数开始处理 `for` 中的初始化表达式。`for` 的初始化部分只执行一次，并不构成循环的一部分。随后，程序指针被提前到初始化表达式结束的分号之后，这个地址赋给了 `temp`。这个位置是条件表达式的开始。然后，进入一个无限的循环，检查循环的条件部分，并将增量表达式的起始地址赋给 `temp2`。然后，找到循环开始的代码。最后，如果条件表达式为 `true`，则解释循环代码块。否则，找到代码块的结尾，继续执行在 `for` 循环之后的语句。假定循环执行，当对 `interp()` 的调用返回时，增加表达式被求值，这个过程重复。当然，如果在循环代码块中遇到了 `break` 语句，这个过程结束。

9.6.12 处理 `cin` 和 `cout` 语句

由于通过 `cin` 和 `cout` 的 I/O 是 C++ 的基础部分，因此 Mini C++ 必须支持它们。然而，Mini C++ 并没有用商业化编译器处理 `cin` 和 `cout` 的方式来处理 I/O 链接。您知道，`cin` 和 `cout` 是预定义的标识符，这两个标识符对应于与标准输入和标准输出相链接的流。通过使用 I/O 运算符 “<<”

和“>>”，它们可以在控制台输入和输出信息。因此，“<<”以及“>>”为了I/O而重载。然而，Mini C++不支持运算符重载。事实上，为了使得Mini C++尽可能的简单，甚至没有支持“<<”和“>>”移位运算符(然而，`get_token()`能够识别这些运算符)。尽管有这些限制，但是解释`cin`和`cout`语句相当容易。

通过`cout`的控制台输出由`exec_cout()`函数处理，代码如下所示：

```
// Execute a cout statement.
void exec_cout()
{
    int val;

    get_token();
    if(*token != LS) throw InterpExc(SYNTAX);

    do {
        get_token();

        if(token_type==STRING) {
            // Output a string.
            cout << token;
        }
        else if(token_type == NUMBER ||
                token_type == IDENTIFIER) {
            // Output a number.
            putback();
            eval_exp(val);
            cout << val;
        }
        else if(*token == '\\') {
            // Output a character constant.
            putback();
            eval_exp(val);
            cout << (char) val;
        }

        get_token();
    } while(*token == LS);

    if(*token != ';') throw InterpExc(SEMI_EXPECTED);
}
```

当遇到`cout`标识符时，会读取下一个令牌。如果下一个令牌不是“<<”，就会报告语法错误。否则，就会进入一个循环，然后输出“<<”右边的字符串或者表达式的值。这个过程持续到`cout`语句的结尾。

`cin`语句由`exec_cin()`函数处理，代码如下所示：

```
// Execute a cin statement.
void exec_cin()
{
    int val;
```

```

char chval;
token_ireps vtype;

get_token();
if(*token != RS) throw InterpExc(SYNTAX);

do {
    get_token();
    if(token_type != IDENTIFIER)
        throw InterpExc(NOT_VAR);

    vtype = find_var_type(token);

    if(vtype == CHAR) {
        cin >> chval;
        assign_var(token, chval);
    }
    else if(vtype == INT) {
        cin >> val;
        assign_var(token, val);
    }

    get_token();
} while(*token == RS);

if(*token != ';') throw InterpExc(SEMI_EXPECTED);
}

```

当遇到 `cin` 标识符时，会读取下一个令牌。如果下一个令牌不是“>>”，则会报告语法错误。否则，就会进入一个循环，获取接收输入的变量，从控制台读取输入，并将这个输入存储在变量中。注意，判断整型或者字符型数据的变量类型被读取。这个过程持续到 `cin` 语句的结尾。

9.7 Mini C++的库函数

由于 Mini C++ 执行的程序不会被编译并链接，因此它使用的任何库例程都必须被 Mini C++ 处理。为此，最好的方法是创建一个接口函数，当遇到库函数时 Mini C++ 就调用这个接口函数。接口函数创建对实际的库函数的调用，并处理所有的返回值。

由于空间的限制，Mini C++ 只支持 4 个“库”函数：`getchar()`、`putchar()`、`abs()` 以及 `rand()`。这些函数被转换为同名库函数的调用。Mini C++ 库例程在文件 `libcpp.cpp` 中。如下所示：

```

// ***** Internal Library Functions *****

// Add more of your own, here.

#include <iostream>
#include <cstdlib>
#include <cstdio>
#include "mcommon.h"

```

```
using namespace std;
```

```
// Read a character from the console.  
// If your compiler supplies an unbuffered  
// character input function, feel free to  
// substitute it for the call to cin.get().
```

```
int call_getchar()  
{  
    char ch;  
  
    ch = getchar();  
  
    // Advance past (  
    get_token();  
    if(*token != '(')  
        throw InterpExc(PAREN_EXPECTED);  
  
    get_token();  
    if(*token != ')')  
        throw InterpExc(PAREN_EXPECTED);  
  
    return ch;  
}
```

```
// Write a character to the display.
```

```
int call_putchar()  
{  
    int value;  
  
    eval_exp(value);  
  
    putchar(value);  
  
    return value;  
}
```

```
// Return absolute value.
```

```
int call_abs()  
{  
    int val;  
  
    eval_exp(val);  
  
    val = abs(val);  
  
    return val;  
}
```

```
// Return a random integer.
```

```
int call_rand()  
{
```

```

// Advance past ()
get_token();
if(*token != '(')
    throw InterpExc(PAREN_EXPECTED);

get_token();
if(*token != ')')
    throw InterpExc(PAREN_EXPECTED);

return rand();
}

```

为了添加您选择的更多的库函数，首先将它们的名称和它们接口函数的地址键入到 `intern_func` 数组中(在 `parser.cpp` 中声明)。随后，按照前面所示的函数，创建合适的接口函数。最后，在 `mcommon.h` 中加入它们的原型。

9.8 mcommon.h 头文件

Mini C++的 3 个源文件，`minicpp.cpp`, `parser.cpp` 以及 `libcpp.cpp`，包括头文件 `mcommon.h`，如下所示：

```

// Common declarations used by parser.cpp, minicpp.cpp,
// or libcpp.cpp, or by other files that you might add.
//
const int MAX_T_LEN = 128; // max token length
const int MAX_ID_LEN = 31; // max identifier length
const int PROG_SIZE = 10000; // max program size
const int NUM_PARAMS = 31; // max number of parameters

// Enumeration of token types.
enum tok_types { UNDEFTT, DELIMITER, IDENTIFIER,
                 NUMBER, KEYWORD, TEMP, STRING, BLOCK };

// Enumeration of internal representation of tokens.
enum token_ireps { UNDEFTOK, ARG, CHAR, INT, SWITCH,
                  CASE, IF, ELSE, FOR, DO, WHILE, BREAK,
                  RETURN, COUT, CIN, END };

// Enumeration of two-character operators, such as <=.
enum double_ops { LT=1, LE, GT, GE, EQ, NE, LS, RS, INC, DEC };

// These are the constants used when throwing a
// syntax error exception.
//
// NOTE: SYNTAX is a generic error message used when
// nothing else seems appropriate.
enum error_msg
{ SYNTAX, NO_EXP, NOT_VAR, DUP_VAR, DUP_FUNC,
  SEMI_EXPECTED, UNBAL_BRACES, FUNC_UNDEF,

```

```

    TYPE_EXPECTED, RET_NOCALL, PAREN_EXPECTED,
    WHILE_EXPECTED, QUOTE_EXPECTED, DIV_BY_ZERO,
    BRACE_EXPECTED, COLON_EXPECTED );

extern char *prog; // current location in source code
extern char *p_buf; // points to start of program buffer

extern char token[MAX_T_LEN+1]; // string version of token
extern tok_types token_type; // contains type of token
extern token_ireps tok; // internal representation of token

extern int ret_value; // function return value
extern bool breakfound; // true if break encountered

// Exception class for Mini C++.
class InterpExc {
    error_msg err;
public:
    InterpExc(error_msg e) { err = e; }
    error_msg get_err() { return err; }
};

// Interpreter prototypes.
void prescan();
void decl_global();
void call();
void putback();
void decl_local();
void exec_if();
void find_eob();
void exec_for();
void exec_switch();
void get_params();
void get_args();
void exec_while();
void exec_do();
void exec_cout();
void exec_cin();
void assign_var(char *var_name, int value);
bool load_program(char *p, char *fname);
int find_var(char *s);
void interp();
void func_ret();
char *find_func(char *name);
bool is_var(char *s);
token_ireps find_var_type(char *s);

// Parser prototypes.
void eval_exp(int &value);
void eval_exp0(int &value);
void eval_exp1(int &value);
void eval_exp2(int &value);

```

```

void eval_exp3(int &value);
void eval_exp4(int &value);
void eval_exp5(int &value);
void atom(int &value);
void sntx_err(error_msg error);
void putback();
bool isdelim(char c);
token_ireps look_up(char *s);
int find_var(char *s);
tok_types get_token();
int internal_func(char *s);
bool is_var(char *s);

// "Standard library" prototypes.
int call_getchar();
int call_putchar();
int call_abs();
int call_rand();

```

9.9 编译并链接 Mini C++解释程序

为了使用 Mini C++, 必须编译并链接 minicpp.cpp, parser.cpp 以及 libcpp.cpp。您几乎可以使用任何流行的 C++编译器, 包括 Borland C++和 Visual C++。例如, 对于 Visual C++, 您可以使用如下的命令行:

```
cl -GX minicpp.cpp parser.cpp libcpp.cpp
```

对于 Borland C++, 可以使用如下的命令行:

```
bcc32 minicpp.cpp parser.cpp libcpp.cpp
```

如果您使用了不同的编译器, 只需要简单地遵循它附带的指示。

提示:

对于 Visual C++的比较老的版本, 可能没有给予 Mini C++足够的堆栈空间。您可以使用 /Fsize 选项来增加堆栈。

为了运行一个程序, 需要在命令行的 minicpp 之后指定它的名称。例如, 通过下面的命令行运行名为 test.cpp 的程序:

```
minicpp test.cpp
```

9.10 演示 Mini C++

本节给出了一些 C++程序来演示 Mini C++的特性和功能。第一个程序演示了 Mini C++支持的所有特性:

```
/* Mini C++ Demonstration Program #1.
```



```

    This program demonstrates all features
    of C++ that are recognized by Mini C++.
*/

int i, j; // global vars
char ch;

int main()
{
    int i, j; // local vars

    // Call a "standard library" function.
    cout << "Mini C++ Demo Program.\n\n";

    // Call a programmer-defined function.
    print_alpha();

    cout << "\n";

    // Demonstrate do and for loops.
    cout << "Use loops.\n";
    do {
        cout << "Enter a number (0 to quit): ";
        cin >> i;

        // Demonstrate the if
        if(i < 0 ) {
            cout << "Numbers must be positive, try again.\n";
        }
        else {
            for(j = 0; j <= i; ++j) {
                cout << j << " summed is ";
                .cout << sum(j) << "\n";
            }
        }
    } while(i != 0);

    cout << "\n";

    // Demonstrate the break in a loop.
    cout << "Break from a loop.\n";
    for(i=0; i < 100; i++) {
        cout << i << "\n";
        if(i == 5) {
            cout << "Breaking out of loop.\n";
            break;
        }
    }

    cout << "\n";

```

```

// Demonstrate the switch
cout << "Use a switch.\n";
for(i=0; i < 6; i++) {
    switch(i) {
        case 1: // can stack cases
        case 0:
            cout << "1 or 0\n";
            break;
        case 2:
            cout << "two\n";
            break;
        case 3:
            cout << "three\n";
            break;
        case 4:
            cout << "four\n";
            cout << "4 * 4 is " << 4*4 << "\n";
            break; // this break is optional
        // no case for 5
    }
}
cout << "\n";

cout << "Use a library function to generate "
    << "10 random integers.\n";
for(i=0; i < 10; i++) {
    cout << rand() << " ";
}

cout << "\n";
cout << "Done!\n";

return 0;
}

// Sum the values between 0 and num.
// This function takes a parameter.
int sum(int num)
{
    int running_sum;
    running_sum = 0;

    while(num) {
        running_sum = running_sum + num;
        num--;
    }
    return running_sum;
}

// Print the alphabet.
int print_alpha()
{

```

```

    cout << "This is the alphabet:\n";

    for(ch = 'A'; ch<='Z'; ch++) {
        putchar(ch);
    }
    cout << "\n";

    return 0;
}

```

下面是运行的样本:

Mini C++ Demo Program.

```

This is the alphabet:
ABCDEFGHIJKLMNOPQRSTUVWXYZ

```

Use loops.

Enter a number (0 to quit): 10

```

0 summed is 0
1 summed is 1
2 summed is 3
3 summed is 6
4 summed is 10
5 summed is 15
6 summed is 21
7 summed is 28
8 summed is 36
9 summed is 45
10 summed is 55

```

Enter a number (0 to quit): 0

```

0 summed is 0

```

Break from a loop.

```

0
1
2
3
4
5

```

Breaking out of loop.

Use a switch.

```

1 or 0
1 or 0
two
three
four
4 * 4 is 16

```

Use a library function to generate 10 random integers.

```

130 10982 1090 11656 7117 17595 6415 22948 31126 9004

```

Done!

下面的程序演示了嵌套循环:

```
// Nested loop example.
int main()
{
    int i, j, k;

    for(i = 0; i < 5; i = i + 1) {
        for(j = 0; j < 3; j = j + 1) {
            for(k = 3; k ; k = k - 1) {
                cout << i << ", ";
                cout << j << ", ";
                cout << k << "\n";
            }
        }
    }

    cout << "done";

    return 0;
}
```

这个程序的部分输出如下所示:

```
0, 0, 3
0, 0, 2
0, 0, 1
0, 1, 3
0, 1, 2
0, 1, 1
0, 2, 3
0, 2, 2
0, 2, 1
.
.
.
```

下一个程序练习了赋值运算符:

```
// Assignments as operations.
int main()
{
    int a, b;

    a = b = 5;

    cout << a << " " << b << "\n";

    while(a=a-1) {
        cout << a << " ";
        do {
```

```

        cout << b << " ";
    } while((b=b-1) > -5);
    cout << "\n";
}

return 0;
}

```

这个程序的输出如下所示:

```

5 5
4 5 4 3 2 1 0 -1 -2 -3 -4
3 -5
2 -6
1 -7

```

下一个程序演示了递归函数。其中，函数 `factr()` 用于计算某个数的阶乘。

```

// This program demonstrates a recursive function.

// A recursive function that returns the
// factorial of i.
int factr(int i)
{
    if(i<2) {
        return 1;
    }
    else {
        return i * factr(i-1);
    }
}

int main()
{
    cout << "Factorial of 4 is: ";
    cout << factr(4) << "\n";

    cout << "Factorial of 6 is: ";
    cout << factr(6) << "\n";

    return 0;
}

```

输出如下:

```

Factorial of 4 is: 24
Factorial of 6 is: 720

```

下面的程序完整地演示了函数的参数:

```

// A more rigorous example of function arguments.

int fl(int a, int b)

```

```

{
    int count;

    cout << "Args for f1 are ";
    cout << a << " " << b << "\n";

    count = a;
    do {
        cout << count << " ";
    } while(count=count-1);

    cout << a << " " << b
        << " " << a*b << "\n";

    return a*b;
}

int f2(int a, int x, int y)
{
    cout << "Args for f2 are ";
    cout << a << " " << x << " "
        << y << "\n";
    cout << x / a << " ";
    cout << y*x << "\n";

    return 0;
}

int main()
{
    f2(10, f1(10, 20), 99);

    return 0;
}

```

这个程序的输出如下所示:

```

Args for f1 are 10 20
10 9 8 7 6 5 4 3 2 1 10 20 200
Args for f2 are 10 200 99
20 19800

```

下面的程序练习了各种循环语句:

```

// Exercise the loop statements.
int main()
{
    int a;
    char ch;

    // The while.
    cout << "Enter a number: ";
    cin >> a;

```

```

while(a) {
    cout << a*a << " ";
    --a;
}
cout << "\n";

// The do-while.
cout << "\nEnter characters, 'q' to quit.\n";
do {
    // Use two "standard library" functions.
    ch = getchar();
    putchar(ch);
} while(ch != 'q');
cout << "\n\n";

// the for.
for(a=0; a<10; ++a) {
    cout << a << " ";
}

cout << "\n\nDone!\n";

return 0;
}

```

这是一个运行的样本:

```

Enter a number: 10
100 81 64 49 36 25 16 9 4 1

Enter characters, 'q' to quit.
This is a test. q
This is a test. q

0 1 2 3 4 5 6 7 8 9

Done!

```

注意, 在这个程序的运行中, 内建的库函数 `getchar()` 是行缓存的, 从而只有按下 ENTER 时, `putchar()` 才会显示字符。这种行为是 Mini C++ 调用库函数 `getchar()` 的结果。您知道, 大多数编译器实现的 `getchar()` 都是行缓存的。关键是内建的、Mini C++ 函数展示了与底层的库函数相同的行为。

最后一个程序演示了嵌套作用域的使用。在这个程序中, 变量 `x` 声明了 3 次: 第 1 次是作为全局变量, 第 2 次是作为 `if` 代码块的局部变量, 第 3 次是在 `while` 代码块中声明。3 个变量是独立的, 彼此各不相同。

```

// Demonstrate nested scopes.

int x; // global x

int main()

```

```

{
    int i;

    i = 4;

    x = 99; // global x is 99

    if(i == 4) {
        int x; // local x
        int num; // local to if statement

        x = i * 2;
        cout << "Outer local x before loop: "
              << x << "\n";

        while(x--) {
            int x; // another local x

            x = 18;
            cout << "Inner local x: " << x << "\n";
        }

        cout << "Outer local x after loop: "
              << x << "\n";
    }

    // Can't refer to num here because it is local
    // to the preceding if block.
    // num = 10;

    cout << "Global x: " << x << "\n";
}

```

输出显示如下:

```

Outer local x before loop: 8
Inner local x: 18
Inner local x: 18
Inner local x: 18
Inner local x: 18
Inner local x: 18
Inner local x: 18
Inner local x: 18
Inner local x: 18
Outer local x after loop: -1
Global x: 99

```

9.11 改进 Mini C++

在设计 Mini C++时考虑了操作的透明性。目的是以最小的代价开发一个易于理解的解释程

序。这个解释程序还是以最容易扩展的方式设计的。因此, Mini C++不是特别快速或者高效。然而, 这个解释程序的基本结构是正确的, 可以使用如下步骤增加执行速度。

事实上所有的商业化解释程序都扩展了预扫描程序的任务。被解释的整个源程序从人类可以理解的方式转化为内部格式。在这种内部格式中, 除了被引用的字符串和常量之外, 所有的内容都转换为整型令牌, 类似于 Mini C++将 C++的关键字转换为整型令牌的方式。Mini C++执行大量的字符串比较时, 就可能这样做。例如, 每次寻找变量或者函数时, 都会发生多次的字符串比较。字符串的比较非常耗时, 然而, 如果源程序中的每个标识符都转换为整型数, 就可以使用比较快速的整型数比较。通常, 源程序到内部格式的转换是一个最重要的转换。可以将其应用于 Mini C++来提高它的效率。速度大为增加。

另一个改进的地方是为变量和函数查找例程, 对于大程序尤其有意义。即使您将这些条目转换为整型令牌, 当前搜索它们的方法也取决于有序地搜索。然而, 可以用其他比较快速的方法来取代。例如, 可以尝试使用 map 容器, 或者可以使用某种散列方法或者树结构。

如前所述, 相对于完整的 C++语法, Mini C++的一个限制是控制语句的实现, 如 if, 必须是用花括号包含的一个代码块。这样做的原因是它大大地简化了 find_eob()函数, 这个函数在控制语句执行之后, 寻找代码块的结尾。find_eob()函数只需要寻找与代码块开始的左花括号匹配的右花括号。您可能会发现, 删除这个限制很有趣。

9.12' 扩展 Mini C++

有两个一般的领域, 可以扩展并增强 Mini C++解释程序: C++特性和辅助特性。下面的部分将就此进行简单讨论。

9.12.1 添加新的 C++特性

可以向 Mini C++添加两类基本的语句。第一类是附加转向语句, 如 goto 和 continue 语句。您可能还想向 switch 中添加对 default 语句的支持。如果学习了 Mini C++解释其他语句的方式, 添加这些特性几乎不会遇到麻烦。如果某些事物在第一时间没有运行, 则可以通过显示处理过的每个令牌的内容来发现问题。

可以添加的第二类语句是对附加数据类型的支持。Mini C++已经为附加的数据类型包含了基本的“钩子”。例如, var_type 结构已经为这种类型的变量包含了一个字段。为了加入其他内建的类型(例如, float, double 以及 long), 只需要将这个值字段的大小提高到您想要保存的最大元素的长度。

添加类是一个更大的挑战。首先, 必须定义实例化对象的方式。为此, 必须分配足够大的内存来保存类的数据成员, 并且将这块内存的引用存储到另一个新的字段, 需要在 var_type 中添加这个字段。还需要处理 public 和 private 的概念。

对指针的支持并不比其他数据类型的支持更加困难。然而, 需要在表达式解析器中加入对指针运算符的支持。一旦实现了指针, 数组就变得容易了。任何数组的空间都应该使用 new 来动态分配, 为此, 指向数组的指针应该被存储在一个新的字段中, 这个字段应该添加到 var_type 中。

为了处理函数的不同返回类型, 需要使用 func_type 结构中的 ret_type 字段。这个字段定义

了函数返回的数据的类型。该字段目前被设定，但是还未使用。

加入对`#include`的支持是比较容易的。这个预处理程序指示可以很容易地在预扫描过程中处理。

最后一点：如果您想要试验语言的结构，不要害怕添加非 C++ 的扩展。例如，您可以很容易地添加第 4 章讨论的 `foreach` 循环。

9.12.2 添加辅助特性

解释程序给了您添加一些有趣并有用的特性的机会。例如，可以加入跟踪工具，来显示每个执行中的令牌。还可以加入在程序执行时，显示每个变量内容的功能。另一个您想加入的特性是一个完整的编辑器，从而可以“编辑并运行”，而不是使用其他编辑器来创建您的 C++ 程序。

了函数返回的数据的类型。该字段目前被设定，但是还未使用。

加入对`#include`的支持是比较容易的。这个预处理程序指示可以很容易地在预扫描过程中处理。

最后一点：如果您想要试验语言的结构，不要害怕添加非 C++ 的扩展。例如，您可以很容易地添加第 4 章讨论的 `foreach` 循环。

9.12.2 添加辅助特性

解释程序给了您添加一些有趣并有用的特性的机会。例如，可以加入跟踪工具，来显示每个执行中的令牌。还可以加入在程序执行时，显示每个变量内容的功能。另一个您想加入的特性是一个完整的编辑器，从而可以“编辑并运行”，而不是使用其他编辑器来创建您的 C++ 程序。